Agnus Azevedo Horta

Implementação de Simulador de Fluidodinâmica em Meso-Escala com o Método DPD

69/2014

CAMPINAS
2014

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA

Agnus Azevedo Horta

Implementação de Simulador de Fluidodinâmica em Meso-Escala com o Método DPD

Dissertação de Mestrado apresentada à Faculdade de Engenharia Mecânica da Universidade Estadual de Campinas como parte dos requisitos exigidos para obtenção do título de Mestre em Engenharia Mecânica, na Área de Mecânica dos Sólidos e Projeto Mecânico

Orientador: Prof. Dr. Luiz Otávio Saraiva Ferreira

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO DEFENDIDA PELO ALUNO Agnus Azevedo Horta, E ORIENTADO PELO PROF. DR. Luiz Otávio Saraiva Ferreira.

.....
ASSINATURA DO ORIENTADOR

CAMPINAS
2014

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

H788i

Horta, Agnus Azevedo, 1967-
Implementação de simulador de fluidodinâmica em meso-escala com o método DPD / Agnus Azevedo Horta. – Campinas, SP : [s.n.], 2014.

Orientador: Luiz Otávio Saraiva Ferreira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Mecânica.

1. Fluidodinâmica computacional (CFD). 2. Simulação por computador. 3. Partículas. 4. Dinâmica da partícula. 5. Sistemas dinâmicos diferenciais. 6. Dinâmica dos fluidos. I. Ferreira, Luiz Otávio Saraiva,1956-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Mecânica. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Implementation of fluid dynamics simulator in meso-scale with the DPD method
Palavras-chave em inglês:
Computational fluid dynamics (CFD)
Computer simulation
Particles
Dynamics of particle
Differential dynamical systems
Fluid dynamics
Área de concentração: Mecânica dos Sólidos e Projeto Mecânico
Titulação: Mestre em Engenharia Mecânica
Banca examinadora:
Luiz Otávio Saraiva Ferreira [Orientador]
Renato Pavanello
Wu Shin Ting
Data de defesa: 29-04-2014
Programa de Pós-Graduação: Engenharia Mecânica

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA MECÂNICA

COMISSÃO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA

DEPARTAMENTO DE MECÂNICA COMPUTACIONAL

DISSERTAÇÃO DE MESTRADO ACADÊMICO

Implementação de Simulador de Fluidodinâmica em Meso-Escala com o Método DPD

Autor: Agnus Azevedo Horta
Orientador: Luiz Otávio Saraiva Ferreira

A Banca Examinadora composta pelos membros abaixo aprovou esta Dissertação:

Prof. Dr. Luiz Otávio Saraiva Ferreira, Presidente
DMC/FEM/UNICAMP

Prof. Dr. Renato Pavanello
DMC/FEM/UNICAMP

Profa. Dra. Wu Shin-Ting
DCA/FEEC/UNICAMP

Campinas, 29 de Abril de 2014.

Dedicatória

A minha Avó Áurea, aos meus pais Agnus e Maria Auxiliadora, a minha esposa Carla e ao meu filho Antônio.

Agradecimentos

Aos meus Amigos, Danilo e Wendell;

Ao meu orientador, Prof. Dr. Luiz Otávio;

Aos membros das bancas de qualificação e defesa, Profa. Dra. Wu-Shin Ting e Prof. Dr. Renato Pavanello;

A Elisabeth, pelo zelo constante;

A Bruna, pela paciência;

A toda minha família;

Aos amigos e companheiros de laboratório e a todos os funcionários da FEM.

The highest education is that which does not merely give us information but makes our life in harmony with all existence.

Rabindranath Tagore

Resumo

HORTA, Agnus Azevedo. Implementação de Simulador de Fluidodinâmica em Meso-Escala com o Método DPD. 2014. 158p. Dissertação (Mestrado). Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Campinas.

Este trabalho teve como objetivo principal a implementação do motor de simulação de um framework de simulação com partículas. O motor de simulação utilizou o método DPD (Dissipative Particle Dynamics), e foi baseado no paradigma de programação orientada a objeto (POO) e no uso de estruturas de dados otimizadas. O motor de simulação foi escrito em linguagem C++. A concepção do sistema foi realizada de forma a facilitar e promover a reutilização e manutenção do código. Buscou-se, também, a flexibilidade e generalização através do uso da linguagem Python na geração dos arquivos de entrada correspondentes a distribuição espacial das partículas, sendo utilizada a linguagem de marcação XML (eXtensible Markup Language) na estruturação dos arquivos resultantes da simulação. No final, o motor de simulação foi avaliado aplicando o problema do fluxo de um fluído entre placas paralelas e os resultados foram validados por comparação com aqueles publicados na literatura e com aqueles obtidos usando-se o simulador Hoomd-Blue.

Palavras-chave: FLuidodinâmica computacional (CFD), Simulação por Computador, Partículas, Dinâmica da partícula, Sistemas dinâmicos diferenciais, Dinâmica dos fluídos.

Abstract

HORTA, Agnus Azevedo. Implementation of Fuid Dynamics Simulator in Meso-Scale with the DPD method. 2014. 158p. Dissertation (Master). Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Campinas.

This work aims the implementation of a simulation engine of a simulation framework with particles, the simulation engine uses the DPD (Dissipative Particle Dynamics) method, based on the paradigm of object oriented programming (OOP) and use of optimized data structures. The simulation engine was written in C + + language. The system design was performed in order to facilitate and promote the reuse and maintainability of the code. Also, we sought flexibility and generalization through the use of Python in the generation of the corresponding input files of the spatial distribution of particles, and (eXtensible Markup Language) XML markup was used for structuring the files resulting from the simulation. At the end, the simulation engine was validated by applying it to the problem of flow of a fluid between parallel plates and the results were validated by comparing then to results from litterature and to results from the Hoomd-Blue simulator.

Keywords:Computational fluid dynamics (CFD), Computer Simulation, Particles, Dynamics of particle, Differential dynamical systems, Fluid dynamics.

Lista de Ilustrações

Lista de Tabelas

Lista de Algoritmos

SUMÁRIO

1 Introdução

Uma das áreas da tecnologia que tem apresentado rápido crescimento é a de sistemas *Lab-On-A-Chip*, com enfase especial nas aplicações de diagnóstico, *Micro-Total-Analysis Systems* (μTAS), onde a maioria dos dispositivos contém vários passos de processamento de fluidos em micro e nano escalas, processamento esse denominado de micro e nanofluídica. A Simulação por Fluidinâmica Computacional *CFD* é uma ferramenta indispensável no fluxo de projeto de sistemas microfluidicos. Muitos dispositivos μTAS não manipulam apenas fluidos simples, mas células em suspensão ou micro-esferas, lidando com situações complexas de fluxo de fluidos, com grande número de objetos em suspensão que influenciam suas propriedades, mas que não podem ser representados como um fluido reológico [1] com propriedades parametrizáveis.(STEINER, 2009)

Os fenômenos físicos e químicos que podem ser estudados se extendem sobre muitas ordens de magnitudes diferentes. Eles podem ser encontrados das menores até as maiores magnitudes de comprimento, indo da investigação da matéria pela mecânica quântica até o estudo da forma do universo. A faixa de dimensões que ocorrem vão da faixa do nanometro (10^{-9} metros) no estudo das propriedades da matéria no nível molecular, até o nível de 10^{23} metros no estudo dos aglomerados de galáxias. Similarmente, as escalas de tempo as quais ocorrem nestes modelos (isto é, o intervalo de tempo típico no qual o fenômeno observado toma lugar) são amplamente diferentes. Elas vão nos exemplos mencionados de 10^{-12} ou mesmo 10^{-15} segundos a 10^{17} segundos. (GRIEBEL *e outros*, 2007)

A figura a seguir ilustra as escalas de comprimento em relação as diferentes escalas de tempo.

[1] Reologia é o estudo do fluxo de matéria, primeiramente no estado líquido, mas também como um 'sólido mole' ou sólidos sob condições no qual ele responde como um flúido plástico em vez de deformar elasticamente em resposta as forças aplicadas. *wikipédia*

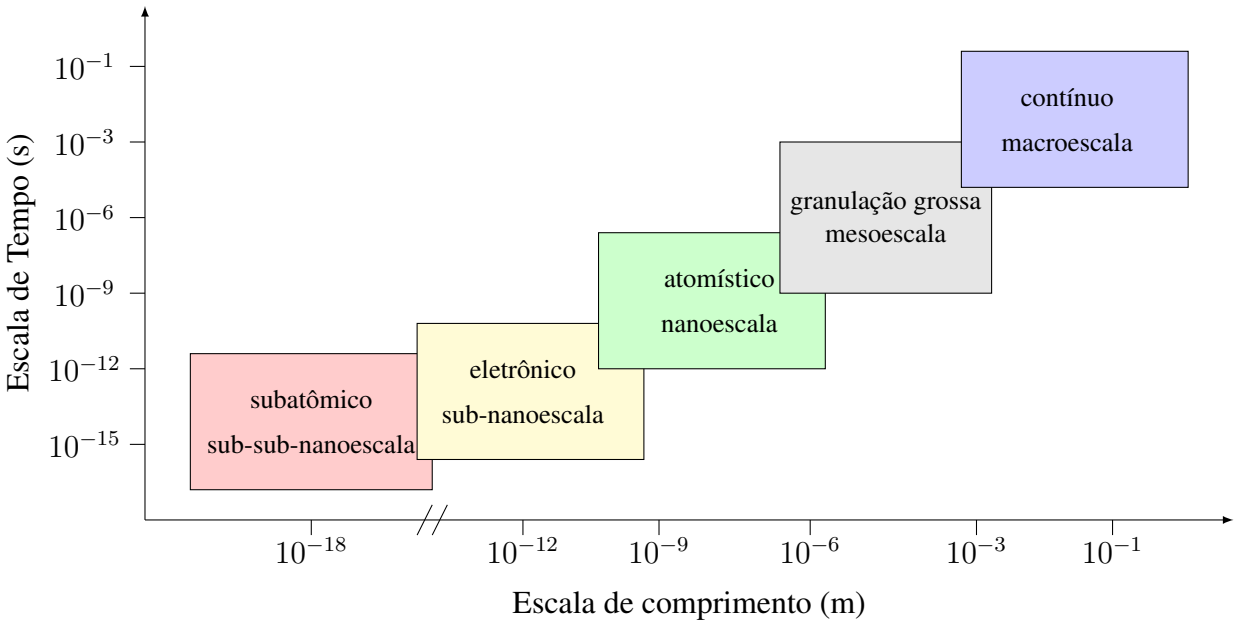

Figura 1.1: Comparação entre as escalas de tempo e comprimento, com base na representação esquemática apresentada por Jonathan E. Mueller e Jacob (2013)

A simulação com modelos de partículas é uma importante área da simulação numérica. Nesses modelos os sistemas físicos constituem-se de partículas discretas e suas interações. Essas partículas possuem as propriedades dos objetos físicos, como por exemplo massa, posição, velocidade ou carga. (GRIEBEL *e outros*, 2007)

Um método de simulação com partículas que simula com sucesso sistemas em equilíbrio é o método conhecido como Dinâmica Molecular (MD), que soluciona as equações de movimento de Newton de maneira a obter a trajetória do sistema em intervalos de tempos discretos. (SCHILLER, 2005)

A simulação de sistemas fluídicos exige muito tempo de CPU e muita memória, o que é um fator limitante para o tamanho do sistema a ser simulado, além de que, alguns fenômenos em fluídos complexos aparecem em escalas de tempo maiores do que as do movimento das partículas individuais. A fim de tratar com estas limitações, uma das técnicas utilizada é a redução dos graus de liberdade, que é obtida pela representação do sistema através de um conjunto de variáveis relevantes que envolvam uma escala de tempo adequada. Em geral o procedimento para redução é chamado de granulação grossa.(SCHILLER, 2005)

Liu (LIU *e outros*, 2007) afirma que o método *Molecular Dynamics (MD)* é uma maneira fundamental de se compreender melhor o comportamento de fluídos em microcanais, mas seu elevado custo computacional torna proibitiva sua aplicação a problemas práticos. Um método mais recente, o *Dissipative Particle Dynamics (DPD)*, proposto por Hoogerbrugge e Koelman em 1992, permite simulações de sistemas mesoscópicos que envolvem interações fluído-estrutura, soluções coloidais, e problemas multi-escala, como também demonstrado por outros autores (KOUMOUTSAKOS; PALMA *e outros*, 2005; 2006). Em função dessas características, o *DPD* tem sido aplicado com sucesso na modelagem numérica de sistemas microfluídicos de elevada complexidade como, por exemplo, a microbomba coloidal publicada por De Palma em 2006 (PALMA *e outros*, 2006), a simulação de fluxo multi-fase em redes de microcanais publicada por Liu em 2007 (LIU *e outros*, 2007), e um microssistema de imuno-ensaios baseado em micro-esferas e uma câmara microfluídica publicado por Steiner em 2009 (STEINER, 2009).

O método DPD é baseado na Mecânica Estatística (ESPAÑOL E WARREN, 1995) e os métodos clássicos, como por exemplo o MD, são baseados em Mecânica Newtoniana (GRIEBEL *e outros*, 2007).

1.1 Objetivo

O objetivo deste trabalho foi desenvolver o motor de simulação computacional de dinâmica dos fluídos que utilize o método *Dissipative Particle Dynamics*, tendo-se em vista seu uso no projeto de dispositivos microfluídicos.

1.2 Justificativa

Em trabalho anterior (HORTA E FILHO, 2012) foram testados alguns dos principais simuladores DPD disponíveis e observou-se que não são apropriados para problemas de engenharia, como por exemplo o microreator de biodisel apresentado (HORTA E FILHO, 2012), por não disporem das condições de fronteira apropriadas e outros recursos tais como recirculação de partículas de fluído, fontes e sumidouros de pressão constante, vazão constante, etc.

Uma das principais linhas de pesquisa desenvolvidas no Laboratório de Supercomputação,

criado e coordenado por nosso orientador, é o estudo de dispositivos microfluídicos. Neste contexto, uma ferramenta insdispensável para o estudo de dispositivos microfluídicos é a simulação computacional, com este fim um dos projetos em andamento é a construção de um framework de simulação, cujo diagrama é apresentado na figura 1.2.

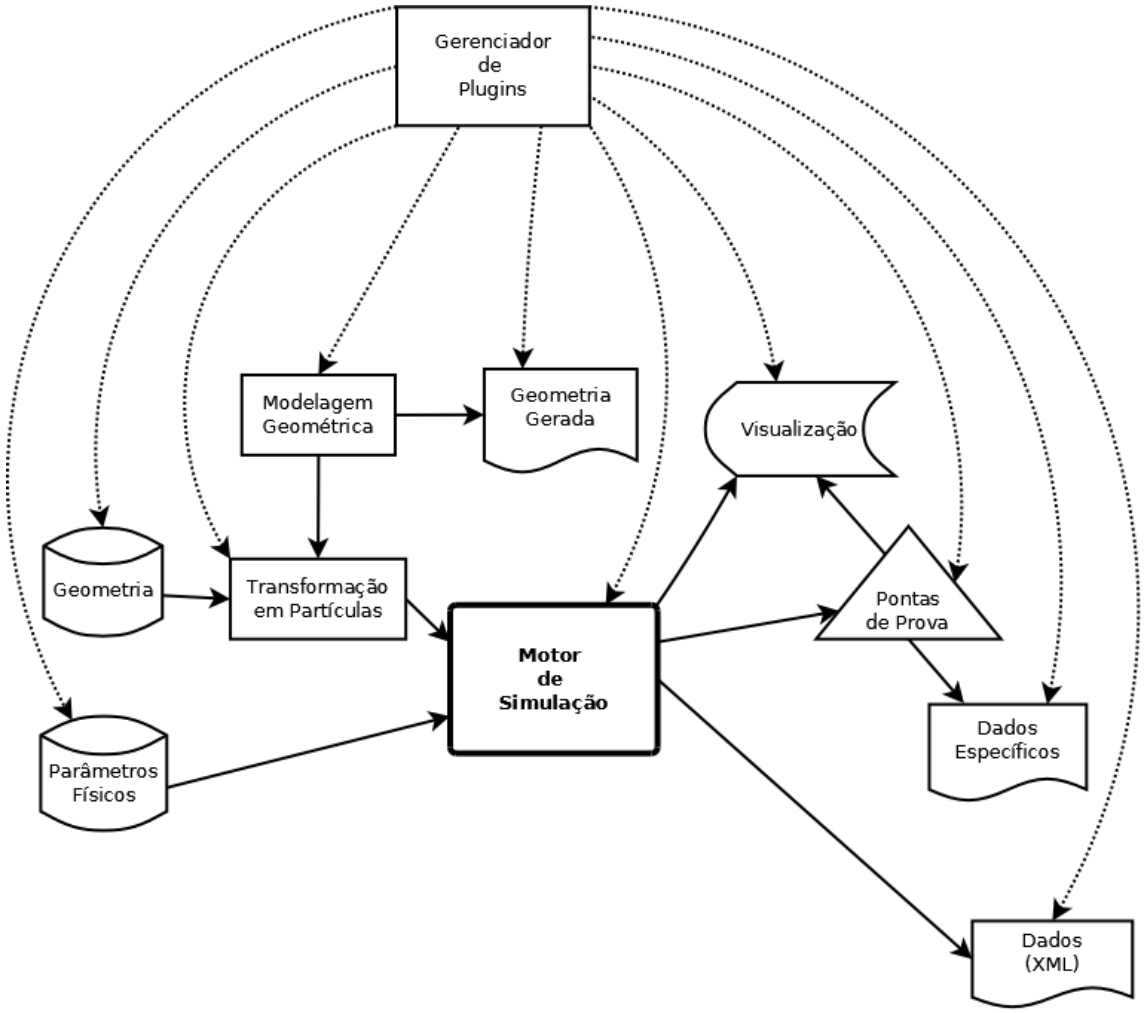

Figura 1.2: Framework de simulação.

O motor de simulação proposto é a peça mais importante do framework, pois é o motor que simula a dinâmica dos fluídos. O framework permitirá a entrada de dados através de arquivos contendo a descrição geométrica dos dispositívos e os parâmetros físicos correspondentes, uma segunda possibilidade é a construção da geometria utilizando das ferramentas de modelagem geométrica disponíveis no framework. A geometria fornecida será transformada em partículas que juntamente com os parâmetros físicos serão processados pelo motor de simulação. O motor de si-

mulação irá gerar os dados que poderão ser visualizados, adquiridos através de pontas de prova (*probes*) ou ainda gravados em arquivos que poderão ser, por exemplo, no formato *XML*. A visualização poderá ser realizada através da integração com ferramentas conceituadas existentes, como o VMD (Visual Molecular Dynamics) ou ParaView, e até mesmo, através da implementação de rotinas que façam uso de bibliotecas como a OpenGL (Open Graphics Library) ou o VTK (Visualization Toolkit). O framework prevê o uso do gerenciamento de extensões de maneira que a capacidade de cada módulo seja expandida, com a incorporação de novos formatos de entrada ou saída de dados, modelagem, visualização e novos métodos de simulação, permitindo desta forma a expansão das funcionalidades já existentes ou a incorporação de novas, possibilitando sua adequação as mais diferentes necessidades.

Como demonstrado por Steiner (2009), é possível simular de forma fidedigna sistemas microfluídicos complexos utilizando o método DPD, e as ferramentas tradicionais de simulação de fluidos não estão aptas a simular os fenômenos de meso-escala presentes em muitos sistemas microfluidicos. Portanto o método DPD foi escolhido para o motor do simulador implementado.

Embora o custo computacional do método DPD seja menor que o do MD, ainda é elevado em comparação com CFD. (STEINER, 2009)

O problema do custo computacional do método DPD pode ser resolvido de duas formas: com o uso de clusters de CPUs, como realizado por Steiner (2009), ou com o uso de GPUs, como realizado por Anderson (2013). Essa última opção mostrou-se, através do simulador Hoomd-blue, superior ao uso de clusters, pois uma única GPU oferece desempenho equivalente a um cluster com 128 a 164 núcleos. (ANDERSON, 2013)

Embora a implementação do método DPD em GPUs possa ter superioridade de desempenho, foi decidido que neste trabalho a implementação do motor de simulação seria realizada em CPU, para que o domínio da programação em GPU não competisse com o domínio dos complexos algoritmos do método.

1.3 Revisão Bibliográfica

A simulação por computador tornou-se uma ferramenta indispensável de investigação e predição dos processos físicos e químicos, permitindo o uso de modelos cada vez mais realistas e complexos. (GRIEBEL *e outros*, 2007)

Uma das primeiras simulações a ser realizada foi a simulação numérica de líquidos densos realizada por Metropolis, Rosenbluth, Rosenbluth, Teller e Teller utilizando o método Metrópolis Monte Carlo (MC), posteriormente chamado de Método Monte Carlo em função do uso intensivo de números aleatórios gerados por computador. Em 1956, Alder e Wainwright utilizando o método Dinâmica Molecular (MD) estudaram a dinâmica de um conjunto de esferas rígidas. Em 1964, as simulação de um líquido real (argônio) foi realizada por Rahman utilizando o método MD (FRENKEL E SMIT, 2001)

As esferas estão sujeitas a uma força de interação, que na sua forma mais usada é gerada pelo potencial de Lennard-Jones, e obedecem as leis de Newton. (STEINER, 2009)

Hoogerbrugge e Koelman (1992) propuseram um método para simulações de fenômenos hidrodinâmicos chamado de *Dissipative Particle Dynamics* que combinava aspectos da dinâmica molecular e a aproximação do *Gás de Rede - Lattice Gas*. Posteriormente Español e Warren (1995) relacionaram a temperatura do sistema ao modelo através de pequenas modificações nos algoritmos originais.

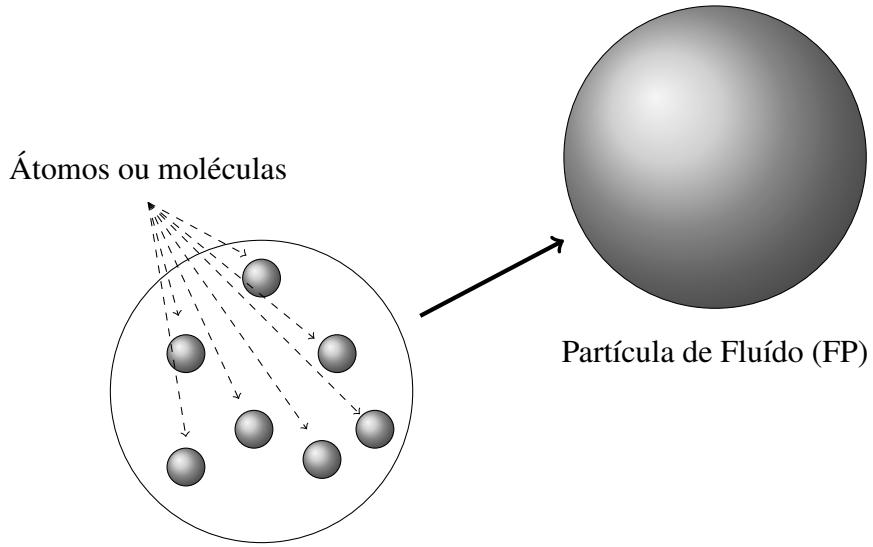

Figura 1.3: Repesentação do processo de granulação grossa (*coarse grain*)

A figura 1.3 ilustra o processo de simplificação. As partículas de fluído (FPs) são representações simplificadas de aglomerados de átomos ou moléculas (GROOT E WARREN, 1997) usadas nas simulações em meso-escala (granulação grossa).

As condições de fronteira são um dos aspectos mais importantes das simulações de interesse das engenharias. No caso das simulações com partículas, há o problema das partículas do fluído penetrarem nas paredes sólidas, que pode ser evitado reinjetando-as no fluído com base em um dos três modelos de reflexão: especular, Maxwelliana e "Bounce-back"(REVENGA *e outros*, 1998).

Posteriormente observou-se, considerando o coeficiente adimensional de atrito e a temperatura, que para pequenos valores de coeficiente de atrito, as reflexões especular e Maxwelliana apresentaram o deslizamento nas paredes, enquanto que a reflexão "bounce back" apresentou um comportamento anômalo da temperatura (REVENGA *e outros*, 1999). O deslizamento apresentado na proximidade das paredes ocorre em função da agitação térmica. Para que ele não ocorra, propõe-se que uma camada de partículas seja colocada fora do domínio da simulação, a uma distância igual ao raio de corte utilizado (WILLEMSEN *e outros*, 2000). Isso apresenta um custo computacional na modelagem de superfícies complexas, que pode ser reduzido implementando-se uma estrutura de parede composta por duas camadas de partículas "congeladas" (com posições fixas), cuja distância das fronteiras são proporcionais ao raio de corte. Quando ocorre de uma partícula entrar nas camadas da parede a ela é aplicado o "bounce back" (DUONG-HONG *e outros*, 2004).

A representção de diferentes sistemas físicos pode ser realizada através do procedimento granulação grossa

1.3.1 Principais Simuladores que usam o Método DPD

A seguir são relacionados alguns dos simuladores existentes atualmente que possibilitam a realização de simulações pelo método DPD.

O sistema de simulação Gromacs (VAN DER SPOEL *e outros*, 2001), que teve seu desenvolvimento iniciado no grupo Herman Berendsen's, do departamento de Química Biofísica da Universidade de Groningen, permite a realização de simulações de dinâmica molecular e foi inicialmente projetado para simulações com biomoléculas como proteínas, lipídios e ácidos nucleicos. Possui uma versão modificada chamada de Gromacs-DPD que implementa o método DPD. O simulador Fluidix, pode ser utilizado em modelos baseados em partículas, apresentando funcionalidades para simulações MD (*Molecular Dinamics*), SPH (*Smoothed-Particle Hydrodynamics*) e DPD. O simulador de propósito geral para dinâmica molecular Hoomd-Blue, cujo desenvolvimento é liderado pelo Grupo Glotzer da Universidade de Michigan, realiza simulações utilizando o método DPD através do componente adicional desenvolvido por Phillips *e outros* (2011). O ambiente de modelagem de simulação Material Studio, distribuído pela empresa Accelrys, concebido para simulação e modelagem de materiais, possui um módulo para simulação DPD. O DL_MESO é um pacote de simulação em escala mesoscópica desenvolvido pelo Dr. Michael Seaton do Departamento de Computação Científica do Laboratório de Daresbury. O LAMMPS, simulador de dinâmica molecular clássica cujo grupo principal é composto por pesquisadores do Laboratório Sandia, também possui um componente que permite realizar simulações com o método DPD através da plataforma MAPS (Materials Processes and Simulations) desenvolvida pela Scieconomics. O ESPResSo (Extensible Simulation Package for Research on Soft matter) é um pacote que permite a execução e análise de simulações de dinâmica molecular. É desenvolvido no Laboratório de Física Computacional da Universidade de Stuttgart, contando com colaboradores de vários países. O DPDmacs é um simulador de dinâmica molecular *coarse-grained* que utiliza o método DPD e foi desenvolvido pelos pesquisadores Michael Patra da Universidade Técnica de Berlim e Mikko Karttunen do Laboratório de Química da Universidade de Warteloo.

Como dito na seção "Justificativa", esses simuladores não são apropriados para problemas de

engenharia por não disporem das condições de fronteira e outros recursos indispensáveis, gerando assim a necessidade da implementação de um simulador apropriado para engenharia.

A partir da revisão bibliográfica, foram selecionados os diversos métodos computacionais utilizados no desenvolvimento do motor de simulação, que serão apresentados no próximo capítulo.

2 Materiais e Métodos

2.1 Simulação por Computador

Segundo Griebel *e outros* (2007) a mera observação de um processo natural não garante a identificação de seus princípios fundamentais. Quando possível, um experimento conduzido por um cientista, possibilita identificar o quanto o evento observado depende das condições escolhidas, permitindo inferências sobre os princípios fundamentais e comportamentos do sistema observado, tendo como objetivo a sua formulação matemática, que geralmente faz uso de equações diferenciais e integrais, as quais codificam a descrição do sistema ou processo, sendo referenciadas como um modelo matemático. Quando confirmados, os modelos não só permitem a descrição precisa do processo observado como também de processos físicos similares considerados certos limites. Os fenômenos geralmente observados em química e física, sob esta abordagem, se estendem em diversas gamas de magnitude, as dimensões podem variar de 10^{-9} metros, no caso do estudo das propriedades da matéria até 10^{23} metros, no caso do estudo dos aglomerados de galáxias.

Busca-se modelar os processos complexos existentes na natureza da maneira mais precisa possível (GRIEBEL *e outros*, 2007). A figura 2.1 apresenta de uma maneira esquemática a abordagem típica de uma simulação numérica, o passo inicial é a definição do fenômeno físico que se deseja simular, um modelo matemático é construido com base nas observações realizadas do fenômeno, este modelo matemático é formado por um sistema de equações diferencias geralmente dependentes do tempo. O modelo matemático geralmente está definido no domínio contínuo, dependendo da abordagem a na próxima etapa é realizada a aproximação algébrica do modelo matemático, com base discretização obtida é desenvolvido um algoritmo computacional para solução do modelo discreto, o qual é implementado e com os parâmetros de entrada os resultados são calculados. Os resultados obtidos na execução dos algoritmos são avaliados de maneira a se estimar o erro cometido pela aproximação, em função disso pode-se buscar uma melhora dos resultados através de uma mudança de algum aspecto da discretização. Os parâmetros utlizados são avaliados, e se necessário o modelo podem ser revisto a fim de se obter melhores resultados. Com base nos resultados experimentais novos aspectos podem ser inferidos. As tarefas realizadas podem ser analisadas com o intuito de que, quando possível, possam ser realizadas em paralelo.

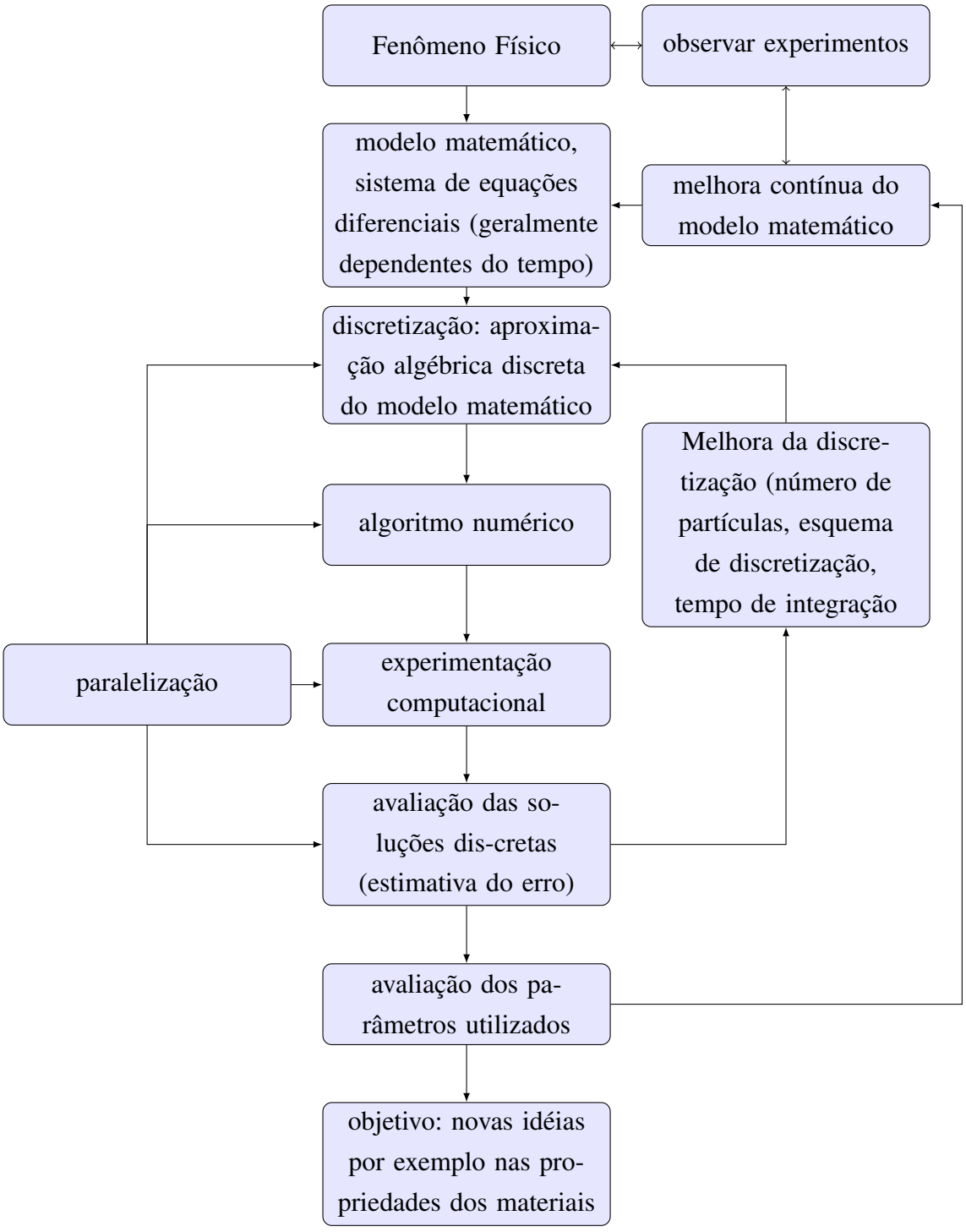

Figura 2.1: Representação esquemática da abordagem típica de uma simulação numérica. (GRIEBEL *e outros*, 2007)

No próximo tópico será apresentado o método DPD, usando na implementação do motor de simulação de sistemas microfluídicos.

2.2 Método DPD

2.2.1 Simulação com o Método DPD

As equações e os métodos de integração numérica usados do método DPD são apresentados nesta seção.

Segundo Steiner (2009), desde sua formulação o método *Dissipative Particle Dynamics* (DPD) vem sendo aprimorado e tornou-se um método poderoso para simulações que envolvem fluxo de fluído e sistema de matéria mole. É baseado na Mecânica Clássica e na Mecânica Estatística. A explicação do método a partir da Mecânica Estatística (ESPAÑOL E WARREN, 1995) é muito complexa, e por isso não será abordada neste trabalho.

Equação do movimento, Termostatos e Equação de Estado

O método DPD representa a dinâmica do fluído pelo comportamento estatístico de um conjunto de átomos/moléculas que constituem as partículas de fluído((HOOGERBRUGGE E KOELMAN, 1992), (ESPAÑOL, 2004)). Essas partículas podem ser irrotacionais ou rotacionais((MARTYS E MOUNTAIN, 1999), (ZHANG *e outros*, 2011)). A compressibilidade do fluído é determinada por parâmetros do modelo((PAN *e outros*, 2013)).

Em DPD, os fluídos e matéria mole são representados pelos pares de *Partículas de Fluido - Fluid Particle* (*FP*) que interagem seguindo a equação de movimento de Newton (STEINER, 2009)

$$\mathbf{F}_i = m_i \ddot{\mathbf{r}}_i \tag{2.1}$$

Na Equação (2.1) m_i é a massa e r_i representa a coordenada espacial da *FP i* no sistema de referência fixo do laboratório, sendo a força total agindo na FP dada por $\mathbf{F}_i$. Nesta forma simplificada $\mathbf{F}_i$ é a soma das parcelas conservativa $\mathbf{F}^C$, dissipativa $\mathbf{F}^D$, aleatórias $\mathbf{F}^R$ e a força externa $\mathbf{F}^{ext}$

$$\mathbf{F}_i = \sum_j (\mathbf{F}_{ij}^C + \mathbf{F}_{ij}^D + \mathbf{F}_{ij}^R \times \sqrt{\Delta t} + \mathbf{F}_{ij}^{ext}) \tag{2.2}$$

que interagem para distâncias menores ou iguais a um raio de corte r_c, e $\mathbf{r}_i$ é a posição da partícula. A força conservativa no DPD padrão representa uma repulsão leve.

A figura 2.2 apresenta a interação entre duas partículas de fluído que estão a uma distância menor que o raio de corte, observar que são consideradas as projeções do vetor velocidade de cada partícula sobre o eixo que passa por elas.

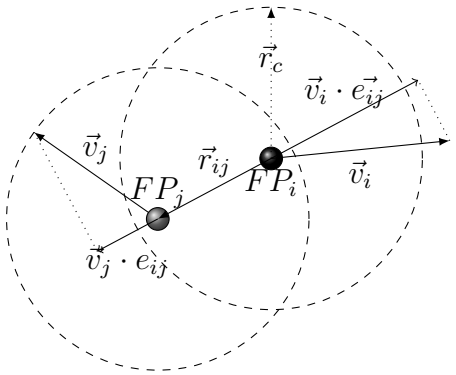

Figura 2.2: Interação entre duas FP. (STEINER, 2009)

O gráfico apresentado na figura 2.3 representa a força de interação em função da distância entre duas partículas, sendo igual a zero para distâncias a partir do raio de corte e atingindo um máximo pré-determinado em função dos parâmetros do modelo, como apresentado na equação 2.3.

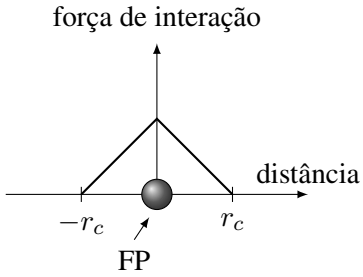

Figura 2.3: Força de interação entre duas FP. (STEINER, 2009)

A força que age entre as partículas *i* e *j* é dada por:

$$\mathbf{F}_{ij}^{C} = \begin{cases} A_{ij} w^{C}\left(r_{ij}\right) \mathbf{e}_{ij} & r_{ij} < 1 \\ 0 & r_{ij} \geq 1 \end{cases} \tag{2.3}$$

onde $\mathbf{r_{ij}} = \mathbf{r}_i - \mathbf{r}_j$, $r_{ij} = |\mathbf{r}_{ij}|$ e $\mathbf{e}_{ij} = \mathbf{r}_{ij}/r_{ij}$ (vide 2.2). A função peso $w^{C}(r_{ij}) = (1 - r_{ij}/rc)$ torna-se zero para uma distância inter-partículas r_{ij} maior que o raio de corte r_c (vide 2.3).

$$\mathbf{F}_{ij}^{D} = \begin{cases} -\gamma w^{D}\left(r_{ij}\right)\left(\mathbf{v}_{ij} \cdot \mathbf{e}_{ij}\right)\mathbf{e}_{ij} & r_{ij} < 1 \\ 0 & r_{ij} \geq 1 \end{cases} \tag{2.4}$$

$$\mathbf{F}_{ij}^{R} = \begin{cases} \sigma w^{R}\left(r_{ij}\right) \xi_{ij} \mathbf{e}_{ij} & r_{ij} < 1 \\ 0 & r_{ij} \geq 1 \end{cases} \tag{2.5}$$

Segundo Español e Warren (1995):

$$\omega^{R}(r_{ij}) = \left(\omega^{D}(r_{ij})\right)^{1/2} \tag{2.6}$$

$$\sigma = \left(2\gamma K_b T\right)^{1/2} \tag{2.7}$$

No próximo tópico serão apresentadas as condições de fronteira periódica do método DPD.

2.2.2 Condições de Fronteira

Neste tópico serão apresentados as condições de fronteira periódica, de pressão e de reflexão.

Assumimos que o domínio da simulação é retangular, isto é, $\Omega = [0,L_1] \times [0,L_2] \times [0,L_3]$, com lados L_1, L_2, L_3 respectivamente. De acordo com as característica de um determinado problema certas condições são impostas na fronteira. Condições periódicas são utilizadas também em problemas não periódicos em função do tamanho limitado do domínio da simulação numérica Ω. Nestes casos, o sistema é extendido artificialmente pela continuação periódica de todo espaço $\mathbb{R}^3$, vide figura 2.4. As partículas que deixam o domínio de um lado reentram no domínio no lado oposto, também as partículas localizadas próximas aos lados opostos do domínio interagem entre si. (GRIEBEL *e outros*, 2007)

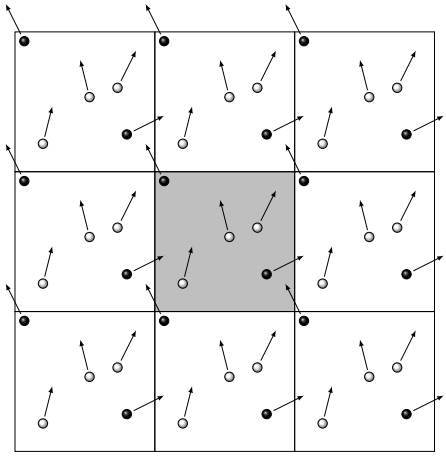

Figura 2.4: Domínio de simulação periódico.
(GRIEBEL *e outros*, 2007)

Em DPD se aplica a mesma idéia de condições de fronteira que em MD.

Em MD as condições de fronteira reflexiva correspondem ao caso de uma caixa de simulação fechada, a partícula que chega a uma distância menor que um certo limite da parede está sujeita a uma força repulsiva que faz com que a partícula seja refletida pela parede, vide figura 2.5. (GRIEBEL *e outros*, 2007)

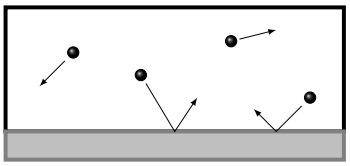

Figura 2.5: Condições de reflexão na fronteira.
(GRIEBEL *e outros*, 2007)

Condições de Reflexão

Em um nível mesoscópico, as paredes representadas devem ser impenetráveis, produzirem o efeito de não deslizamento e não devem afetar as propriedades do fluído que esta sendo simulado. (VISSER *e outros*, 2005).

As paredes sólidas podem ser modeladas como uma fronteira virtual (VB), fronteira de parede congelada (FWB) e fronteira de parede fina (SWB). A fronteira virtual é obtida pela modificação das condições da fronteira sem modelar fronteiras físicas, neste método as partículas que escapam de uma fronteira reaparecem na fronteira oposta com velocidade e posição modificadas, não sendo adequado para fronteiras complexas e não tem sido utilizado ultimamente. A fronteira de parede congelada é o tipo de modelagem mais amplamente utililzado, em geral são utilizadas uma ou mais camadas de partículas congeladas em um padrão ou dispostas aleatoreamente fora da região do fluído, uma versão da FWB modificada chamada de fronteira determinada por força (FCB) é obtida pelo uso de várias camadas de partículas congeladas e do uso de um parâmetro de repulsão. O método SWB consiste de uma única camada de partículas imóveis exatamente na fronteira. (RANJITH *e outros*, 2013)

A impenetrabilidade é reproduzida pela reintrodução das partículas que atravessam a parede de volta ao domínio do fluído, o não deslizamento é obtido em geral pela inversão da direção da velocidade. Em geral este tipo de implementação causa alterações na densidade em regiões próximas a parede e distorções nas propriedades do fluído. (RANJITH *e outros*, 2013)

Na reflexão *bounce-back*, vide figura 2.6, ambos os componentes da velocidade são invertidos. (REVENGA *e outros*, 1999)

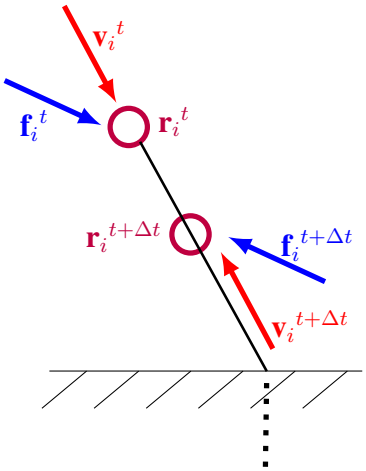

Figura 2.6: Reflexão para trás.
(RANJITH *e outros*, 2013)

Na reflexão especular, vide figura 2.7, a componente paralela do *momentum* da partícula é conservado e a componente normal invertida. A colisão da partícula de fluído com a parede é considerada como sendo elástica e especular como mostrado na figura 2.7. A componente especular da reflexão é calculada como apresentado a na equação 2.8. (REVENGA *e outros*, 1999)

$$\mathbf{d}_s = 2(\mathbf{d}_n \cdot \mathbf{d}_i)\mathbf{d}_n - \mathbf{d}_i \tag{2.8}$$

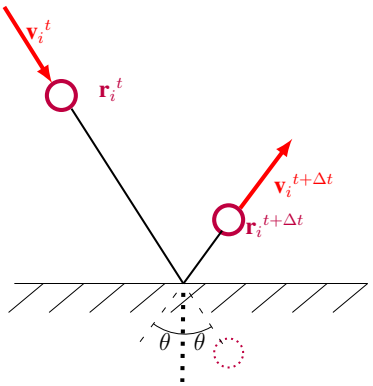

Figura 2.7: Reflexão Especular.
(RANJITH *e outros*, 2013)

Na reflexão *bounce-forward*, vide figura 2.8, ambos os componentes da velocidade são in-

vertidos. (REVENGA *e outros*, 1999)

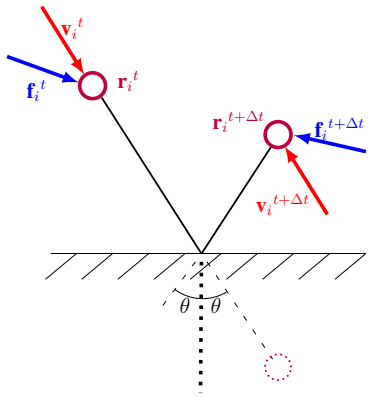

Figura 2.8: Reflexão para frente.
(RANJITH *e outros*, 2013)

O método de reflexão para frente apresenta a mesma condição de não deslizamento que o método de reflexão para trás sem apresentar uma acumulação não realística de partículas nas cavidades. (VISSER *e outros*, 2005)

A seguir será apresentada a condição de fronteira de pressão.

Condições de fronteira de pressão

É necessário aplicar condições de fronteira apropriadas que atuem no fluído de maneira a fazê-lo fluir pela estrutura microfluídica. Diferentes estratégias desenvolvidas para MD podem ser aplicadas também para o DPD: o método do reservatório, o gravitacional e da fronteira reflectiva. Embora o MD e o DPD tenham algumas similaridades deve-se levar em conta algumas diferenças essenciais, como por exemplo a alta compressibilidade dos fluídos DPD, quando do uso de métodos que implementam um certa condição de fronteira.(STEINER, 2009).

Em nosso trabalho consideramos uso do método gravitacional que apresenta um bom compromisso entre eficiência computacional e ajustabilidade da pressão.(STEINER, 2009)

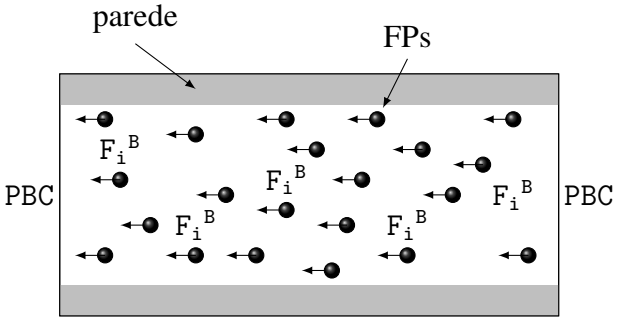

Figura 2.9: Método de força gravitacional.
(STEINER, 2009)

No *método gravitacional*, um campo de força constante é aplicado a todas as *FPs* no sistema. Existindo apenas um parâmetro a ser definido, a força de corpo *FB*, a qual possui uma relação linear de um para um com o gradiente de pressão, sendo independente da largura do canal segundo experimentos realizados por Steiner (2009).

A temperatura dos sistemas simulados com o DPD tende a permanecer constante em presença de pertubações, devido ao caráter termostático do método, conforme detalhado no próximo tópico.

2.2.3 Termostato

Em experimentos físicos a manutenção da temperatura é obtida através troca de calor entre o sistema em consideração e um sistema significativamente maior, o que é chamado de banho de calor ou termostato. A influência do sistema menor no banho de calor é desprezível, o qual tem sua temperatura considerada constante e pré-determinada. Com o passar do tempo o sistema menor adota a temperatura do banho de calor. Para obter o mesmo efeito em uma simulação o sistema deve perder ou ganhar energia de uma maneira apropriada de forma que a temperatura desejada seja atingida.(GRIEBEL *e outros*, 2007)

À temperatura constante, a energia de um sistema de N partículas flutua. De maneira a simular tal sistema, necessitamos de um mecanismo para introduzir flutuações de energia, isto é realizado utilizando-se forças aleatórias que agem nos átomos e alteram sua energia cinética. (ANDERSEN, 1980)

$$\Omega_{ij} = W(|\mathbf{r}_i - \mathbf{r}_j|)\{\pi_{ij} - \omega(\mathbf{p}_i - \mathbf{p}_j) \cdot \mathbf{e}_{ij}\} \tag{2.9}$$

No caso do DPD, de uma maneira simplificada a componente relativa ao termo estocástico tende a "aquecer" o sistema, enquanto que as demais tendem a relaxar qualquer movimento relativo. Ambas agem juntas como um termostato: se o sistema começa a ficar "quente", o segundo termo de amortecimento (sendo proporcional ao movimento relativo das partículas) irá dominar e esfriar o sistema, ao contrário em sistemas que começam a "esfriar" o termo estocástico irá dominar e fará com que o sistema atinja temperaturas mais altas. O termo de amortecimento é primeiramente responsável pelos efeitos de viscosidade e o termo estocástico leva aos efeitos corretos da pressão.(HOOGERBRUGGE E KOELMAN, 1992)

O par de forças dissipativas e aleatórias agem coletivamente como um termostato e também obrigam a conservação do momentum local, o qual é necessário para a correta descrição das interações hidrodinâmicas de longo alcance. (WANG *e outros*, 2008)

O método DPD não representa diretamente um fluído real. Há necessidade de um mapeamento que estabeleça a correspondência entre o domínio DPD e o domínio físico do fluído, conforme apresentado no próximo tópico.

2.2.4 Mapeamento das unidades DPD para unidades físicas

O mapeamento das simulações para os experimentos reais deve ser realizado com critério, as propriedades do fluído expresso pelo método DPD devem ser consideradas como de um modelo de fluído genérico, não podendo ser transferidas diretamente para um fluído real, como por exemplo a água (STEINER, 2009).

Na determinação de como os parâmetros do modelo influenciam a viscosidade, Steiner (2009) realizou simulações nas quais uma caixa de simulação de dimensões $10r_c \times 10r_c \times 10r_c$ com as condições de fronteira de Lees-Edwards, foi utilizada para determinar a viscosidade η em função dos parâmetros A_{ij}, σ e ρ, conforme gráficos apresentados a seguir:

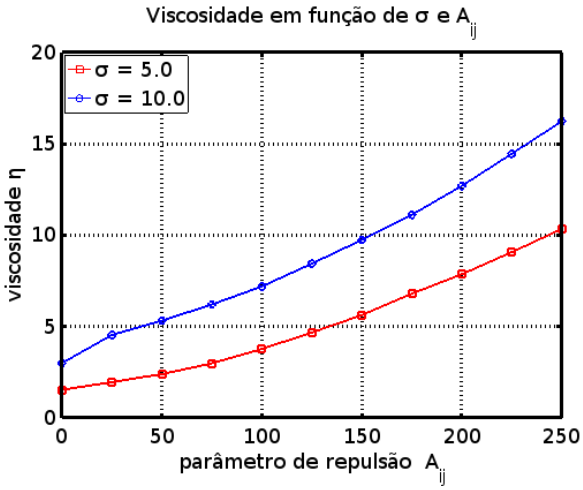

Figura 2.10: Repulsão A_{ij}. (STEINER, 2009)

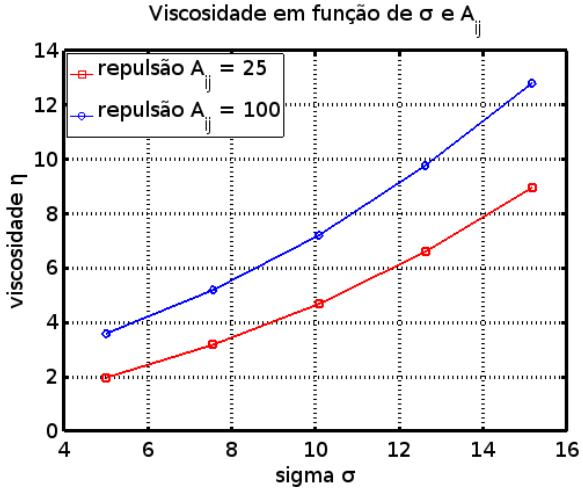

Figura 2.11: Sigma σ. (STEINER, 2009)

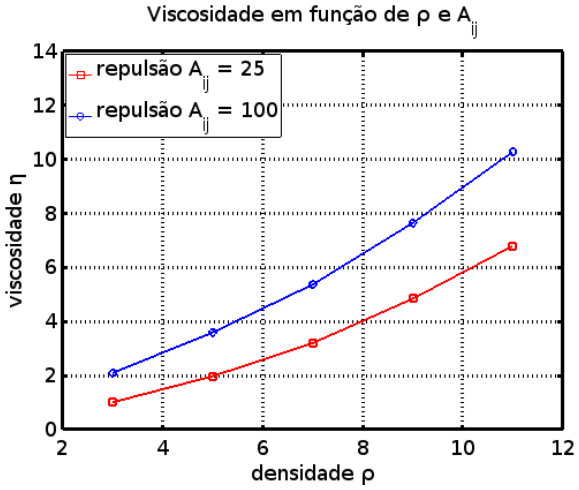

Figura 2.12: Densidade ρ. (STEINER, 2009)

Um outro exemplo de mapeamento é o utilizado por Wang *e outros* (2008) na simulação de um sistema de polímeros [1]. Cada partícula do DPD representa um segmento de *Kuhn* [2] do polímero. O monômero de *Kuhn* do poliestireno (PS) possui $M_0 = 720g/mol$ e um comprimento de *Kuhn* de $b = 18$Å. As cadeias de polímeros simuladas podem ser mapeadas como tendo $r_c \approx 22$Å, $m \approx 720g/mol$ e unidade de tempo $\tau_0 = r_c(m/\epsilon)^{1/2} = 37ps$ onde ϵ é tomado como sendo K_bT

[1] macromoléculas formadas a partir de unidades estruturais menores (os monômeros). *wikipédia*

[2] O comprimento de Kuhn é um tratamento teórico, desenvolvido por Werner Kuhn, no qual um cadeia de um polímero real é considerada como uma coleção de N segmentos de Kuhn cada qual com um comprimento de Kuhn b. *wikipédia*

com $T = 298K$.

Comprimento da cadeia (N)	**Raio de rotação (R_{g0})**	**Coeficiente de Difusão (D_0)**	**Menor tempo de afrouxamento (τ)**
10	1,16	0,068	7,9
20	1,77	0,044	27,7
30	2,26	0,034	54
40	2,68	0,028	99
60	3,45	0,020	180
100	4,7	0,015	510

Tabela 2.1: Propriedades dinâmicas e equilíbrio estático de cadeias de polímeros diluídas em soluções (WANG *e outros*, 2008)

2.3 Algoritmos Utilizados

A implementação computacional das forças aleatórias do método DPD (equações 2.3, 2.4 e 2.5, etc.) depende fortemente de algoritmos de geração de números pseudo-aleatórios, que são apresentados no próximo tópico.

2.3.1 Geração de números pseudo-aleatórios

O método DPD permite construir um modelo de partículas estocástico para sistemas fluídicos (HOOGERBRUGGE E KOELMAN, 1992).

Hoogerbrugge e Koelman (1992) utilizaram uma amostragem de números aleatórios a partir de uma distribuição uniforme em simulações 2D. Neste sentido é interessante verificar se o gerador de números pseudo-aleatórios utilizado é adequado, evitando-se assim resultados não condizentes com o método. De maneira que este tópico tem como objetivo apresentar alguns geradores, certificando de sua corretude e analisando seu desempenho, não pretendendo ser uma discussão aprofundada do assunto por este não ser o foco do trabalho.

A força estocástica é responsável pelo movimento Browniano das partículas do polímero ou colóide que é simulado com o DPD, a qual corresponde aos graus de liberdade que foram eliminados no processo de descrição através de uma granulação grossa (ESPAÑOL, 2005).

Español (2005) consideram o termo ξ_{ij} na equação 2.5 como sendo um ruído gaussiano.

Existem vários geradores de ruído gaussiano, dentre eles consideramos:

- Método do Teorema do Limite Central Donadio (2012);
- Método Polar de Marsaglia (DONADIO, 2012);
- Método Ziggurat (THOMAS *e outros*, 2007).

A seguir apresentaremos os algoritmos correspondentes a cada um dos métodos.

- Método do Teorema do Limite Central apresentado por Donadio (2012);

  No algoritmo do Teorema do Limite Central na linha 1, a variável que conterá o número pseudo aleatório tem o seu valor inicial definido em zero. Na linha 2, é definido um laço de repetição que determina que os comandos a seguir (linhas 3 e 4) sejam executados N vezes, sendo que quanto maior o valor de N mais preciso será o valor gerado, contudo maior será o tempo de processamento. Na linha 3, um valor peseudo aleatório uniformemente distribuido no intervalo $[0,1]$ é gerado e atribuido para variável U. Na linha 5, aplicando o teorema do Limite Central é acumulado o valor pseudo aleatório gerado na linha 4. Na linha 10, o valor gerado é ajustado para que sua média seja igual a zero. Na linha 11, é realizado o ajuste para que a variância correspondente seja igual a 1.

Algoritmo 1 Algoritmo para a geração de um ruido gaussiano (DONADIO, 2012)

$X \leftarrow 0$
para i de 1 até N **faça**
    $U \leftarrow uniform()$
    $X \leftarrow X + U$
fim para
▷ para distribuição uniforme em [0,1], $\mu = 0.5$ e $\sigma = \frac{1}{12}$
▷ ajusta X de maneira que $\mu = 0$ e $\sigma = 1$
$X \leftarrow X - N/2$ ▷ define a média como 0
$X \leftarrow X * sqrt(12/N)$ ▷ ajusta a variância para 1

- Método Polar de Marsaglia

Algoritmo 2 Algoritmo para a geração de um ruido gaussiano pelo Método Polar de Marsaglia(DONADIO, 2012)

Função MPM_RAND($media, variancia$)
    $spareready \leftarrow 0$ ▷ spare e spareready estáticas
    se $spareready$ **então**
        $spareready \leftarrow 0$
        Retorne $spare * variancia + media$
    senão
        Repita
            $u \leftarrow uniform() * 2 - 1$
            $v \leftarrow uniform() * 2 - 1$
            $s \leftarrow u * u + v * v$
        Até $(s < 1 \ \& \ s \neq 0)$
    fim se
    $spare \leftarrow v * sqrt(-2.0 * log(s)/s)$
    $spareready \leftarrow 1$
    Retorne $media + variancia * u * sqrt(-2.0 * log(s)/s)$
fim Função

- Método Ziggurat

Algoritmo 3 Algoritmo para a geração de um ruido gaussiano pelo Método Ziggurat (THOMAS *e outros*, 2007)

Repita
  $i \leftarrow 1+\lfloor nU_1 \rfloor$ ▷ Usualmente n é uma potência de 2: que pode ser obtido pelo deslocamento de bits
  $x \leftarrow x_i U_2$
  se $|x| < x_{i-1}$ **então**
    Retorne z ▷ Ponto completamente dentro do retângulo.
  senão
    se $i \neq n$ **então** ▷ Observar que (x_{i-1}) e (xi) são tabelas de busca.
      $y \leftarrow ((x_{i-1}) - (x_i))U$ ▷ Gera a posição aleatória vertical.
      se $y < ((x) - (x_i))$ **então** ▷ Testa a posição com base na PDF.
        Retorne x ▷ Ponto dentro de uma cunha.
      fim se
    senão
      Retorne $|x| > r$ da cauda
    fim se
  fim se
Até $(1 = 1)$

Aos números pseudo-aleátorios gerados devem ser aplicados testes que os avaliem qualitativamente, indentificando se atendem nossa expectativa no que se refere ao tipo de distribuição ao qual supostamente pertencem, neste sentido a próxima seção apresenta um dos testes que podem ser utilizados.

2.3.2 Teste Kolmogorov-Smirnov

O principal propósito na geração de números pseudo-aleatórios é obter sequências que se comportem como se fossem aleatórias, posteriormente a questão passa a ser como decidir se uma sequência é suficientemente aleatória. Existem vários métodos estatísticos que fornecem medidas quantitativas da aleatoriedade. Em 1933, A. N. Kolmogorov propôs um teste baseado na estatística apresentada na equação 2.10, e em 1939 N. V. Smirnov sugeriu várias modificações no teste, incluindo a análise individual de K_n^+ e K_n^-, as quais parecem ser as estatísticas mais convenientes

para aplicações computacionais que envolvam a determinação da aleatoriedade de uma sequência. (KNUTH, 1997)

$$K_n = \sqrt{n} \max_{-\infty<x<+\infty} |F_n(x) - F(x)| = \max(K_n^+, K_n^-) \tag{2.10}$$

O teste Kolmogorov-Smirnov é utilizado para decidir se uma amostra provêm de uma população com uma determinada distribuição, este teste é baseado na função de distribuição empírica (ECDF). Dado um conjunto valores em ordem crescente, $X_1, X_2, \cdots, X_N$, a CDF é definida como $E_n = n(i)/N$, onde $n(i)$ é a quantidade de valores menores que X_i.(NIST/SEMATECH, 2012)

A figura 2.13 apresenta a função distribuição empírica de um conjunto de 30 valores e a função de distribuição acumulativa (cdf) normal. O test K-S é baseado na máxima distância entre as duas curvas.

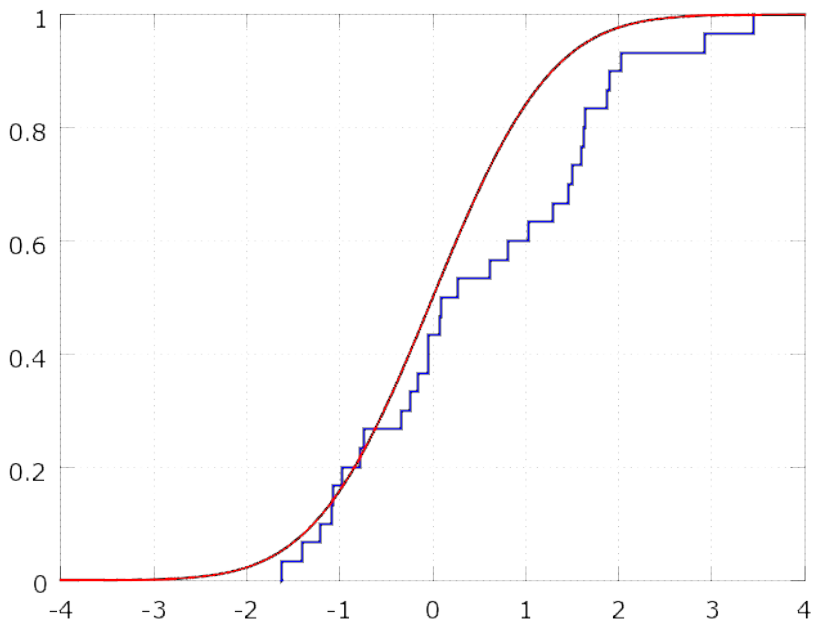

Figura 2.13: Kolmogorov-Sminorv estatística

O teste K-S tem como aspectos positivos o fato de que a estatística em si não depende da cdf em questío, e por ser um teste exato (independe do tamanho da amostra para as aproximações serem válidas). Porém apresenta as seguintes limitações: se aplica somente as distribuições contínuas,

é mais sensível no centro do que nas caudas e a distribuição deve ser totalmente especificada. (NIST/SEMATECH, 2012)

O teste Kolmogorov-Smirnov é definido por: H_0 : os dados seguem a distribuição H_1 : os dados não sequem a distribuição Teste : no qual F é a distribuição acumulativa teórica da distribuição

$$D = \max_{1 \leq i \leq N} \left(F(Y_i) - \frac{i-1}{N}, \frac{i}{N} - F(Y_i) \right) \tag{2.11}$$

Na próxima seção será apresentado o algoritmo de lista de vizinhos para aceleração dos cálculos das interações entre partículas.

2.3.3 Lista de Vizinhos

Em uma simulação com N partículas, nos laços de repetição internos do programa, calcula-se a separação entre uma dada partícula i e cada partícula j. Ignora-se a partícula j caso ela esteja a uma distância maior que o raio de corte r_c, evitando-se cálculos desnecessários. Neste método, o tempo necessário para examinar todos os pares é proporcional a N^2. Para aumentar a velocidade de processamento, Verlet sugeriu que se mantenha uma lista de vizinhos para uma dada partícula, a qual é atualizada a cada intervalo de tempo de maneira que entre os intervalos de tempo o programa irá considerar apenas as partícula presentes na lista, reduzindo assim drasticamente o número de pares de partícula considerados. (ALLEN E TILDESLEY, 1989)

Geração da Lista de Vizinhos

O objetivo do algoritmo de geração da lista de vizinhos é examinar a posição atual de todas as N partículas e construir, para cada uma, a lista daquelas que estío a uma distância menor ou igual ao raio de corte. Isto pode ser feito de maneira trivial comparando-se todas as partículas com todas as demais, o que irá requerer um tempo $O(N^2)$ para sua execução, tornando a simulação para sistemas grandes proibitiva em termos de consumo de tempo. Aprimoramentos no algoritmo reduzem o tempo consumido, na média dos casos, para uma escala de tempo $O(N)$. Para melhorar

o desempenho geral do sistema, códigos de Dinâmica Molecular como o LAMMPS empregam um truque padrão. A distância de corte utilizada na lista de vizinhos é escolhida como sendo r_{max}, maior que o valor de r_{cut} utilizado no cálculo do par de forças. Assim, a lista de vizinhos somente necessita ser atualizada quando uma partícula se moveu a uma distância maior que $\frac{1}{2}(r_{max} - r_{cut})$, o que usualmente ocorre a cada 10 ou mais passos de tempo.

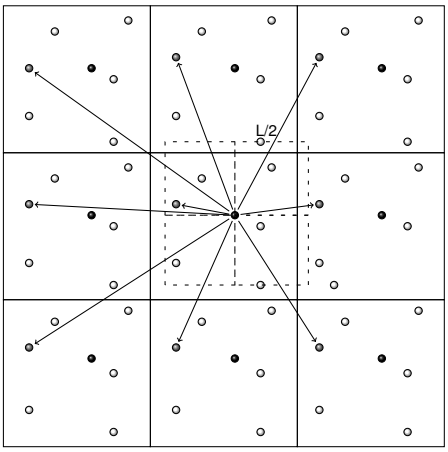

Figura 2.14: Representação esquemática da fronteira periódica com o conceito de distância mínima. (BAYDÍM, 2008)

A convenção da imagem mínima, ilustrada pela 2.14 através do conceito de distância mínima, determina que, em um sistema periódico, o raio de corte mais longo não pode ser maior que a metade da menor dimensão das células que compõe a grade de células. Este requisito é baseado na periodicidade do sistema, de maneira que se uma partícula considera sua própria imagem periódica, forças duplicadas são calculadas e o resultado será incorreto. (VAN DER SPOEL *e outros*, 2001)

Algoritmo 4 Constrói a lista de vizinhos: implementação em CPU (ANDERSON *e outros*, 2008)

para todas as partículas i **faça**
  $\overrightarrow{A} \Leftarrow \overrightarrow{R}_i$
  $A_{cell} \leftarrow$ célula contendo $\overrightarrow{A}$
  $N_{neigh} \leftarrow 0$
  para todas as 27 células C na vizinhança de A_{cell} **faça**
    para todas partículas j contidas na célula C **faça**
      $\overrightarrow{B} \Leftarrow \overrightarrow{R_j}$
      $d\overrightarrow{r} \leftarrow$ minimum image of $\overrightarrow{B} - \overrightarrow{A}$
      se $\mid d_{\overrightarrow{r}} \mid \leq r_{max}$ **então**
        $j \Rightarrow NBL_{N_{neigh},i}$
        $N_{neigh} \leftarrow N_{neigh} + 1$
      fim se
    fim para
  fim para
fim para
$N_{neigh} \Rightarrow NN_i$

2.3.4 Forças de curto alcance

A abordagem de forças de curto alcance explora o fato de que ocorre um rápido decaimento dos potenciais de curto alcance e das forças associadas. Assim, as interações de curto alcance podem ser adequadamente aproximadas apenas considerando a vizinhança geometricamente próxima de cada partícula. (GRIEBEL *e outros*, 2007)

Em cada passo do método de integração utilizado, o cálculo das forças atuantes em cada partícula requer a determinação da interação entre as partículas e a sua correta consideração na somatória das forças. Para as interações que são limitadas somente as partículas vizinhas próximas de uma partícula não faz sentido considerar todas as partículas na somatória das forças. Devem ser consideradas apenas as partículas que irão contribuir para o potencial e para a força, respectivamente. A mesma consideração pode ser feita para o caso de forças e potenciais que decaem rapidamente com a distância. (GRIEBEL *e outros*, 2007)

O par de forças de curto alcance define a força entre qualquer par de partículas em função do vetor de deslocamento que as separa, considerado igual a zero para distâncias maiores que um raio de corte r_{cut}.(ANDERSON *e outros*, 2008)

O fato de se considerar um determinado raio de corte introduz um erro no cálculo das forças que altera levemente a energia total do sistema. Além disso, o potencial e forças correspondentes não são mais contínuos e a energia total do sistema já não é exatamente conservada. Entretanto, se o raio de corte r_{cut} é escolhido grande o bastante, os efeitos resultantes destas descontinuidades é muito pequeno.(GRIEBEL *e outros*, 2007)

O cálculo das forças para todas as N partículas é apresentada no Algoritmo (5). Com base na terceira lei de Newton esta implementação incrementa as forças para as partículas i e j no mesmo passo, reduzindo pela metade o número de operações em ponto flutuante que são necessárias. Deve-se considerar o fato de que com isso aumenta-se o número de acessos à memória, o que por sua natureza aleatória reduz o ganho de desempenho obtido com a redução nos cálculos das forças. (ANDERSON *e outros*, 2008)

Algoritmo 5 Calcula o par de forças: implementação em CPU (ANDERSON *e outros*, 2008)

Requer: $\overrightarrow{F}_k$ deve ser inicializada com $\overrightarrow{0}$ para todos os k

Requer: NBL_{ji} armazena apenas os vizinhos para os quais $i < k$

1: **para** todas as partículas i **faça**
2: $\quad\overrightarrow{A} \Leftarrow \overrightarrow{R}_i$
3: $\quad$**para** $j = 0$ to $NN_i - 1$ **faça**
4: $\quad\quad k \Leftarrow NBL_{ji}$
5: $\quad\quad\overrightarrow{B} \Leftarrow \overrightarrow{R}_k$
6: $\quad\quad d\overrightarrow{r} \leftarrow$ minimum image of $\overrightarrow{B} - \overrightarrow{A}$
7: $\quad\quad$**se** $\mid d_{\overrightarrow{r}} \mid \leq r_{cut}$ **então**
8: $\quad\quad\quad\overrightarrow{C} \leftarrow$ força$(d\overrightarrow{r})$
9: $\quad\quad\quad\overrightarrow{C} + \overrightarrow{F}_i \Rightarrow \overrightarrow{F}_i$
10: $\quad\quad\quad-\overrightarrow{C} + \overrightarrow{F}_k \Rightarrow \overrightarrow{F}_k$
11: $\quad\quad$**fim se**
12: $\quad$**fim para**
13: **fim para**

Inicialmente a estrutura que armazena a somatória das forças para cada partícula é inicializada e considera-se que a estrutura de dados que armazena a lista de vizinhos de cada partícula já foi devidamente alimentada, observa-se que como a partícula i contém a partícula k em sua vizinhança e considerando a aplicação da terceira lei de Newton ...

O algoritmo irá considerar todas as partículas da simulação, armazenado a cada passo a posição da partícula em $\overrightarrow{A}$, e de posse da posição da partícula, percorrerá sua vizinhança identificando o índice de cada uma de suas vizinhas e armazenando a posição correspondente em $\overrightarrow{R}$.

Calcula a distância entre as duas partículas e verifica se essa é menor que o raio de corte, calculando a força de interação entre as duas e acumulando-a para cada partícula.

A próxima seção apresenta a metodologia de integração no tempo das equações de movimento das partículas.

2.3.5 Integração no Tempo das Equações de movimento das Partículas de Fluído

O conceito de discretização implica na transformação da descrição de um problema definido em um intervalo contínuo em uma descrição definida apenas num conjunto finito de pontos. Inicialmente foi utilizada na solução de equações diferenciais, transformando-as em um sistema de equações a diferenças cuja solução se aproxima da solução das equações diferenciais nos pontos de discretização. O uso da discretização simplifica o processo de se calcular os parâmetros das partículas em um novo instante de tempo em função dos parâmetros em instantes de tempo anteriores (GRIEBEL *e outros*, 2007).

Com os modelos de força definidos anteriormente a dinâmica das FPs podem ser calculadas através de vários esquemas de integração. Os três algoritmos mais utilizados são (*Verlet*, *Leap-Frog*, *Velocity-Verlet*) (STEINER, 2009)

Velocity-Verlet

Neste algoritmo (STEINER, 2009), posição, velocidade e aceleração são calculadas no mesmo instante de tempo. O algoritmo poussi 3 passos principais. Incialmente as posições no instante de tempo $(t + \Delta t)$ e a velocidade no instante de tempo intermediário $(t + \frac{1}{2}\Delta t)$ são determinados:

$$\begin{aligned} \mathbf{r}(t + \Delta t) &= \mathbf{r}(t) + \Delta t \mathbf{v}(t) + \tfrac{1}{2}\Delta t^2 \mathbf{a}(t) \\ \mathbf{v}(t + \tfrac{1}{2}\Delta t) &= \mathbf{v}(t) + \tfrac{1}{2}\Delta t)\mathbf{a}(t) \end{aligned} \tag{2.12}$$

No passo seguinte, a força com base na posição no instante de tempo $(t + \Delta t)$ e a velocidade no instante de tempo intermediário são calculadas:

$$\mathbf{f}(t + \Delta t) = \mathbf{f}(\mathbf{r}(t + \Delta t), \mathbf{v}(t + \frac{1}{2}\Delta t)) \tag{2.13}$$

E finalmente, a velocidade no instante intermediário é corrigida para o instante de tempo $(t + \Delta t)$:

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t + \frac{1}{2}\Delta t) + \frac{1}{2}\Delta t \mathbf{a}(t + \Delta t) \tag{2.14}$$

O diagrama apresentado na figura 2.15 ilustra cada uma das etapas do algoritmo Velocity-Verlet. Sendo que as etapas 1, 2 e 3 correspondem ao primeiro passo, 4 ao segundo passo e etapa 5 ao terceiro passo.

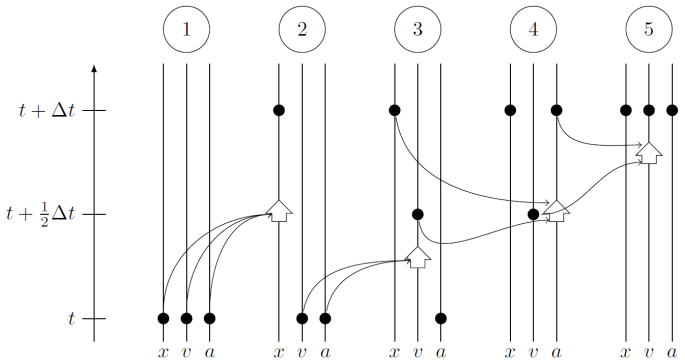

Figura 2.15: Etapas do Velocity-Verlet

O algoritmo de integração Velocity-Verlet para DPD é menos sensível as variações no tamanho do intervalo de tempo que o algoritmo de *Euler* utilizado quando da proposição do método DPD, apresentando uma significante melhora na precisão numérica a um pequeno custo computacional conforme apresentado por Martys e Mountain (1999). Por estas caracteristicas foi o algoritmo utilizado neste trabalho.

Na próxima seção serão apresentadas as principais ferramentas computacionais utilizadas neste trabalho.

2.4 Ferramentas Computacionais Utilizadas

Esta seção apresenta as principais ferramentas computacionais que foram utilizadas em função de alguns dos aspectos do contexto da simulação, mas não diretamente inerentes ao motor de simulação, principalmente em função da validação dos resultados gerados.

O próximo tópico apresenta o visualizador que foi utilizado.

2.4.1 VMD - Visual Molecular Dynamics

O VMD foi projetado por Humphrey *e outros* (1996) para modelagem, visualização e análise de sistemas biológicos como proteínas, ácidos nucléicos, etc. Também pode ser utilizado para

visualizar moléculas mais genéricas, como o VMD pode ler arquivos de banco de dados de proteínas (PDB) e visualizar a estrutura contida. O VMD possui uma ampla gama de méotodos para renderização e colorização de moléculas: pontos, linhas, CPK esferas e cilindros, e outros métodos. O VMD também pode ser utilizado para animar e analizar a trajetória de uma simulação de dinâmica molecular (MD). Em particular, o VMD pode atuar como uma estância final de um programa externo de MD, exibindo e animando as moléculas sob simulação em um computador remoto. (VMD, 2014)

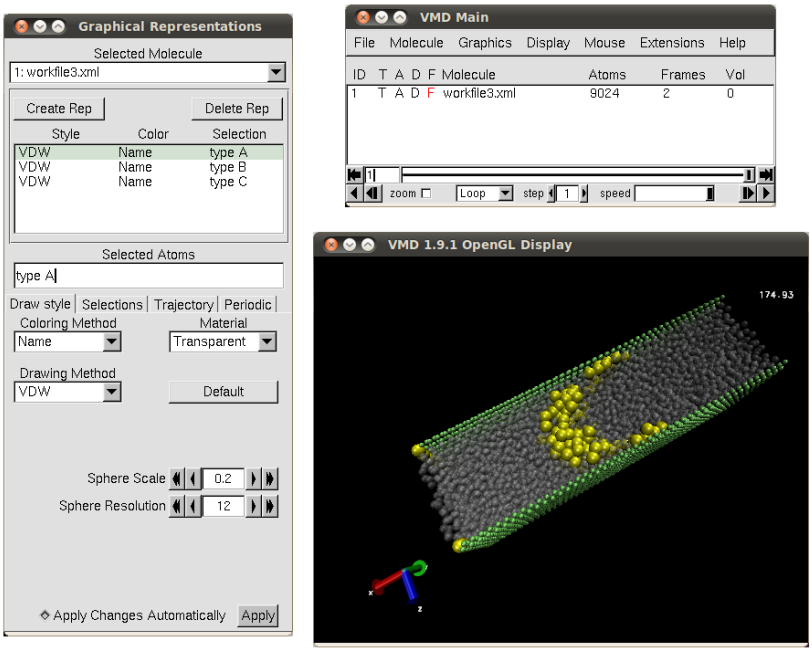

Figura 2.16: Interface do software de visualização VMD

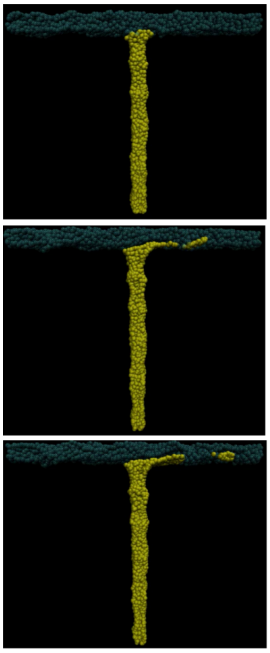

Figura 2.17: Simulação do Microrreator de biodiesel visualizada com o VMD. (HORTA E FILHO, 2012)

Por suas características e funcionalidades o VMD foi adotado com visualizador padrão para o simulador, sendo que a funcionalidade de se conectar com um servidor de dados e exibir em tempo real a simulação é um diferencial deste visualizador, a comunicação com o servidor é realizada através do protocolo IMD, o qual é apresentado na próxima seção.

2.4.2 IMD - Interactive Molecular Dynamics

Interactive Molecular Dynamics (IMD) permite a manipulação de moléculas em simulações de dinâmica molecular com feedback em tempo real e display gráfico. A comunicação entre o programa de visualização (VMD) e o programa de dinâmica molecular (NAMD), executando em uma única máquina ou em múltiplas, é obtida através de um socket de comunicação eficiente.(STONE *e outros*, 2001)

Stone *e outros* (2001) implementaram um protocolo de comunicação que consiste de um pequeno cabeçalho de 16 bytes, seguido de um segmento o qual seu conteúdo e comprimento é especificado pelo cabeçalho. Para máxima eficiência, o cabeçalho e os dados são enviados em uma única operação de escrita do *socket*, idealmente em um único pacote de rede. A interface IMD consiste de um pequeno conjunto de funções em linguagem C que podem ser adaptadas para qualquer programa de dinâmica molecular ou programa de visualização.

Exemplos de algumas funções disponíveis no socket de comunicação:

- vmdsock_listen : coloca o servidor em modo de escuta para receber pedidos de conexío
- vmdsock_write : escreve um determinado conjunto de dados no referido processo de comunição
- vmdsock_read : lê um determinado conjunto de dados do processo de comunição identificado

Exemplos de algumas funções disponíveis no IMD:

- imd_send_energies : envia as energias
- imd_send_fcoords : envia a coordenadas e forças dos átomos
- imd_recv_energies : recebe as energias
- imd_send_energies : recebe as coordenadas dos átomos e respectivas forças

Caso não haja a disponibilidade do VMD e/ou IMD, para que sejam visualizados os resultados da simulação, foi implementado um visualizador utilizando o OpenGl. Na próxima seção apresentaremos o OpenGL.

2.4.3 OpenGL

OpenGl não é uma linguagem de programação, ela é uma Interface de Programação de Aplicativos (API) (WRIGHT *e outros*, 2007). Esta interface consiste de mais de 700 comandos distintos que podem ser utilizados para especificar objetos e as operações necessárias para se produzir aplicativos tridimensionais interativos. (SHREINER, 2010)

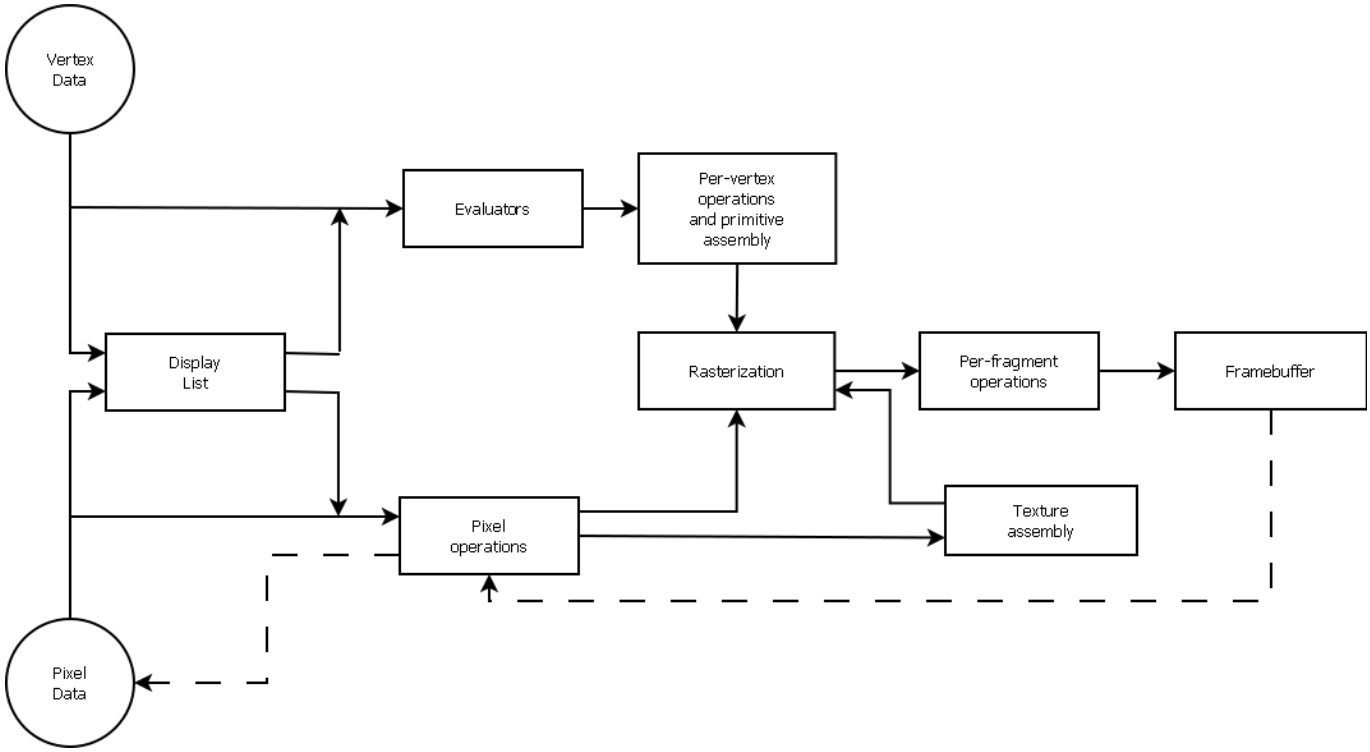

Figura 2.18: OpenGL ordem das operações

2.4.4 OpenMP

OpenMP é um modelo de programação paralelo para multiprocessadores de memória compartilha e memória compartilhada distribuída. Aprimorar o desempenho dos aplicativos computacionais é único propósito prático do processamento paralelo. O principal objetivo do OpenMP é minizar a complexidade introduzida quando utilizamos o paralelismo ao código do aplicativo. O fator pelo qual o tempo utilizado na solução de um problema pode ser aprimorado comparado ao uso de somente um único processador é chamado de *speedup*. (CHANDRA *e outros*, 2001)

Com base no exposto no parágrafo anterior utilizamos na implementação os recursos de paralelismo do OpenMP para agilizar o processamento em algumas tarefas que são simples e não apresentam nenhum aspecto complicador para serem executadas em paralelo, como por exemplo a inicialização do vetor utilizado no cálculo da força de interação entre as partículas.

Considerando que o processo de computacional é dividido em três partes *a*) configuração (*setup*) *b*) computação (*compute*) *c*) finalização (*finalization*)

Mattson *e outros* (2004) apresenta alguns aspectos quantitativos da computação paralela. O tempo total de execução de um programa em um único elemento de processamento (PE), é dado pela soma do tempo para cada uma das partes,

$$T_{total}(1) = T_{setup} + T_{compute} + T_{finalization} \tag{2.15}$$

Considerando que as etapas de *configuração* e *finalização* não podem ser realizadas de maneira concorrete com quaisquer outras atividades, e a atividade de computação pode ser dividida em tarefas que podem ser executadas nas PEs que estío disponíveis, com um número de total de passos computacionais igual a versão original sem concorrência. O tempo total para a realização de toto o processmaneto em P PEs pode ser dado por:

$$T_{total}(P) = T_{setup} + \frac{T_{compute(1)}}{P} + T_{finalization} \tag{2.16}$$

Este cálculo, apresentado na equação 2.16, considera uma situação ideal. Entretanto a idéia que o processamento possui uma parte serial (a qual o aumento de número de PEs, não altera) e uma parte paralelizada (na qual o aumento do número de PEs é significativo, diminuindo o tempo de execução) é realista. Assim este modelo embora simples captura esta importante relação.

Uma relação importante que podemos encontrar é a que expressa quío significativo é o aumento do número de PEs, e indica quío mais rápido um programa será executado, chamada de velocidade relativa S

$$S(P) = \frac{T_{total}(1)}{t_{total}(P)} \tag{2.17}$$

Uma outra medida relacionada é a eficiência E, a qual apresenta a velocidade S normalizada pelo número de PEs

$$E(P) = \frac{S(P)}{P} \tag{2.18}$$

$$E(P) = \frac{T_{total}(1)}{P \times t_{total}(P)} \tag{2.19}$$

Uma situação ideal seria quando a velocidade relativa S for igual a P, o número de PEs. Esta é conhecida como aceleração linear perfeita (*perfect linear speedup*). Infelizmente, este uma situação que raramente pode ser atingida em função dos tempos de *configuração* e *finalização* não são melhorados pela inclusão de novas PEs, limitando a aceleração, pois estas etapas não podem ser realizadas concorrentemente, sendo chamadas de etapas seriais. O seu tempo de execução representa uma fração do tempo total, chamada de fração serial, denotada por γ

$$\gamma = \frac{T_{configurao} + T_{finalizao}}{t_{total}(1)} \tag{2.20}$$

Sendo assim a parte que poder ser paralelizada do programa igual a $(1 - \gamma)$. Reescrevendo a

expressão para o tempo total de computação com P PEs temos:

$$T_{total} = \gamma \times T_{total}(1) + \frac{(1-\gamma) \times T_{total}(1)}{P} \tag{2.21}$$

Reescrevendo S em termos da nova expressão $T_{total}(P)$, obtemos a famosa *Amdahl´s law*

$$S(P) = \frac{T_{total}(1)}{(\gamma + \frac{(1-\gamma)}{P}) \times T_{total}(1)} \tag{2.22}$$

$$S(P) = \frac{1}{\gamma + \frac{(1-\gamma)}{P}} \tag{2.23}$$

Portanto um algoritmo paralelo ideal sem sobrecarga na parte paralelea, a aceleração pode seguir a equação 2.23. O que acontecerá se tornarmos nosso algoritmo paralelo ideal e utilizarmos um grande número de processadores? Tomando o limite como P indo ao infinito nossa expressão S chegará:

$$S(P) = \frac{1}{\gamma} \tag{2.24}$$

A equação 2.24 nos fornece o limite superior para a aceleração obtida em um algoritmo cuja parte serial representa γ da computação total. Este conceito é vital para o projetista de algoritmos. Pode não fazer muito sentido implantar um algoritmo paralelo complexo (e escalonável) se 10% ou mais do algoritmo é serial, sendo que 10% é um percentual muito comum de se encontrar para parte serial de algoritmos.

Neste capítulo foram apresentados os principais métodos e programas computacionais utilizados neste trabalho. O próximo capítulo é dedicado à implementação do motor de simulação com o método DPD.

3 Implementação

Nesse capítulo é apresentada a implementação do motor de simulação. O tópico a seguir traz uma visão geral da lógica do sistema.

3.1 Fluxograma do Sistema

O diagrama da figura 3.1, representa o fluxograma do sistema implementado correspondente ao motor de simulação. Podemos em linhas gerais observar que ele pode ser dividio em três etapas. A primeira etapa corresponde a de entrada de dados, representada pela leitura dos *Parâmetros* e dos *Dados Geométricos*. A segunda etapa corresponde ao *Motor de Simulação*, representado no código pelo método *Integração* (vide anexo C.1). O modulo *Integração*, realiza a construção da *Lista de Partículas*, *Lista de Vizinhos*, primeiro passo do método *Velocity-Verlet* onde são calculadas as novas posições das partículas e a velocidade no instante intermediário, o segundo passo onde são cálculas as *Forças* utilizando as equações do método DPD, e o terceiro passo que corresponde ao cálculo da velocidade das partículas no próximo instante de tempo. A terceira etapa corresponde a exportação dos dados com a gravação da *Velocidade Média* e o arquivo *Dump* no formato XML.

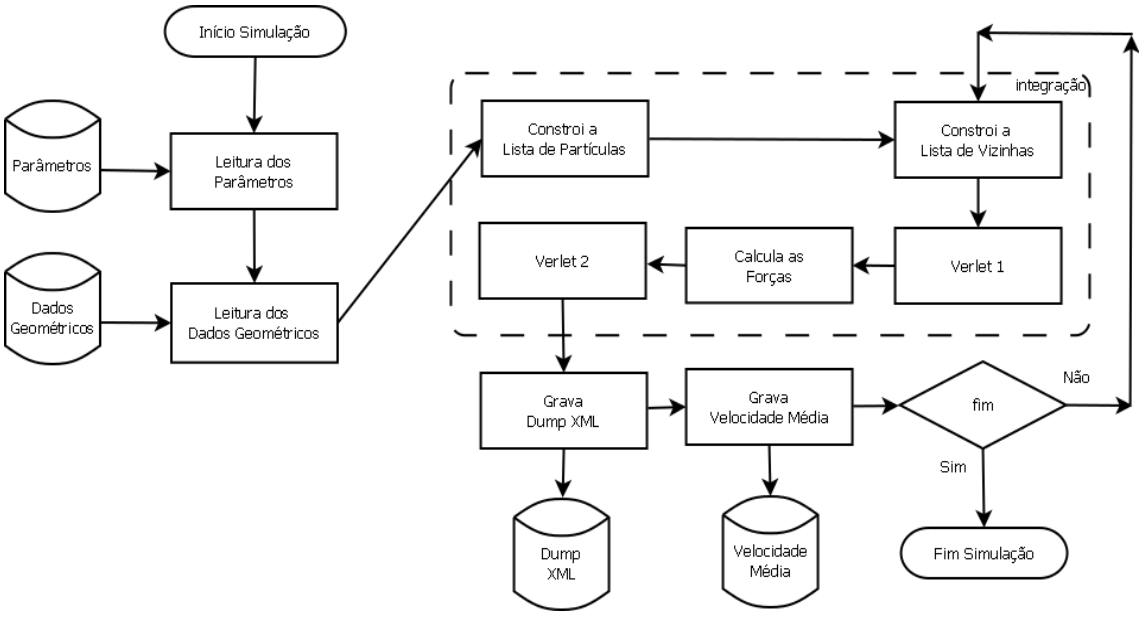

Figura 3.1: Fluxograma do simulador

3.2 Leitura dos Parâmetros

Os parâmetros (físicos, de processamento) da simulação são fornecidos através de um arquivo texto no qual os dados são lidos sequencialmente e sua posição no arquivo define o parâmetro que representa. Na tabela 3.1 são apresentados os respectivos parâmetros e sua posição relativa no arquivo de parâmetros.

Linha	Parâmetro	Descrição
1	use_vmd	especifica se irá habilitar o servidor para o vmd, permitindo a visualização em tempo real da simulação. 1 habilita servidor, 0 desabilita
2	dt	δ_t, intervalo de tempo entre cada iteração da simulação
3	t_end	instante final no qual a simulação encerrará, equivalente em segundos
4	l	dimensão da caixa de simulação, representado por três valores: l_x, l_y e l_z
5	rc	raio de corte a ser considerado para as FPs
6	sigma	0,015
7	epsilon	
8	gm	gama
9	al	alfa
10	al1	alfa1
11	sg	sigma1
12	fb	força externa, representada por três valores fb_x, fb_y e fb_z

Tabela 3.1: Tabela com os parâmetros da simulação

Algoritmo 6 Inicialização da lista de células ligadas

1: ▷ Calcula o número de células correspondentes a cada dimensão, proporcional ao raio de corte
2: $nc[0] \leftarrow \lfloor l[0]/(rc * 1.5) \rfloor$
3: $nc[1] \leftarrow \lfloor l[1]/(rc * 1.5) \rfloor$
4: $nc[2] \leftarrow \lfloor l[2]/(rc * 1.5) \rfloor$
5: $pnc \leftarrow nc[0] * n[c1] * nc[2]$ ▷ Calcula o número total de células
6: $head \leftarrow$ endereço da região de memória para se armazenar pnc números inteiros ▷ Aloca memória para a lista de células e de partículas
7: $ll \leftarrow$ endereço da região de memória para se armazenar qt números inteiros

Inicialização do lista de células e partículas (SistemaParticula::ll_inicializa)

O algoritmo 6 apresenta a inicialização das estruturas de dados para células e lista de partículas. As células devem ter as dimensões no mínimo maiores que o raio de corte para que

o método de células conectadas possa ser aplicado, L_d é a dimensão da *d-ésima* coordenada e nc_d o número de células na *d-ésima* coordenada, de maneira que a relação $r_{cut} \leq L_d/nc_d$ deve ser respeitada. O maior número de células em cada direção será calculado através da equação 3.1. (GRIEBEL *e outros*, 2007)

$$nc_d = \left\lfloor \frac{L_d}{r_{cut}} \right\rfloor \tag{3.1}$$

São calculadas as quantidades de células em cada direção (linhas 2 a 4), e consequentemente o número total de células (linha 6). Sendo alocados dinamicamente dois vetores, um para as células e outro para as partículas (linhas 8 e 9).

Inicialização da lista de partículas vizinhas próximas (SistemaParticula::NBL_inicializa)

Algoritmo 7 Inicialização da lista de partículas vizinhas

1: $NBL \leftarrow dAlloc(rows, cols)$ ▷ Aloca memória para armazenar a matriz que representa a lista de partículas vizinhas
2: $NN \leftarrow$ endereço da região de memória para se armazenar qt números inteiros ▷ Aloca memória para armazenar um vetor que conterá o número de partículas vizinhas de cada partícula

Aloca dinâmicamente uma matriz para armazenar a lista de partículas vizinhas, e um vetor para armazenar a quantidade de vizinhos de cada partícula.

3.3 Leitura dos dados Geométricos

Após a leitura dos parâmetros da simulação o próximo passo será ler os dados relativos a posição, velocidade, tipo e massa de cada partícula, um exemplo do arquivo que contém estes dados pode ser observado no anexo D.1.

O algoritmo 8 apresenta a parte inicial da leitura do arquivo de dados geométricos. Nesta leitura é indentificado o tipo de elemento que está sendo lido. A leitura do arquivo será realizada até que seu fim seja encontrado, linha 1. Na linha 2 é verificado se já esta sendo lido os dados correspondentes a algum elemento. Nas linhas de 4 a 10, identifica-se qual o elemento foi lido,

setando a chave para o valor correspondente. Na linha 12 é verificado se a chave possui um valor correspondente ao término da leitura do elemento que estava sendo lido. A proxima etapa será a leitura dos atributos correspondentes ao elemento em questão.

Algoritmo 8 Leitura dos dados geométricos : elementos

```
1:  Enquanto não for final do arquivo de dados geométricos faça
2:      se ch = 0 então
3:          lê nome do elemento
4:          se nome = "position" então
5:              ch ← 1
6:          senão se nome = "velocity" então
7:              ch ← 2
8:          senão se nome = "mass" então
9:              ch ← 3
10:         senão se nome = "type" então
11:             ch ← 4
12:         senão se ch ≤ 0 então
13:             ch ← 0
14:         fim se
15:     fim se
```

O algoritmo 9 descreve a leitura dos atributos correspondentes a um determinado elemento.

Na linha 16, é identificado se o elemento em questão possui atributos. Possuindo, move-se para o primeiro atributo, linha 17. Da linha 18 até a linha 25 é realizada a leitura dos atributos. Lê-se o atributo, linha 19. Caso o elemento seja *"position"* (ch=1) e o atributo seja *"num"* atribui-se o valor do atributo para a variável que armazena a quantidade de partículas do sistema, linha 21. Na linha 22, com base no número de partículas, inicializa-se o sistema de partículas. Na linha 24, avança-se para o próximo atributo. Pelo laço de repetição podemos observar que apenas o atributo *"num"* do elemento *"position"* é considerado, os demais são ignorados. Na linha 26, avança-se para o próximo elemento.

Algoritmo 9 Leitura dos dados geométricos : atributos

```
16:  se o elemento possui atributos então
17:      move para o primeiro atributo
18:      Repita
19:          lê nome do atributo
20:          se ch = 1 & nome = "num" então
21:              qt ← quantidade de partículas
22:              inicializa o sistema de partículas
23:          fim se
24:          avança para o próximo atributo
25:      Até não tiver mais atributos
26:      move para o elemento;
27:  fim se
```

No algoritmo 10, serão lidos os valores de cada elemento. Na linha 28 é identificado se o elemento possui valores.

Algoritmo 10 Leitura dos dados geométricos : dados

```
28:  se o elemento possui valores então
29:      se ch = 1 então
30:          ch ← −1
31:          carrega a posição inicial de cada partícula
32:      senão se ch = 2 então
33:          ch ← −1
34:          carrega a velocidade inicial de cada partícula
35:      senão se ch = 3 então
36:          ch ← −1
37:          carrega a massa de cada partícula
38:      senão se ch = 4 então
39:          ch ← −1
40:          carrega o tipo de cada partícula
41:      fim se
42:  fim se
43: fim Enquanto
```

Nas linhas 29 a 41, será identificado o tipo de elemento de acordo com a chave anteriormente definida (algoritmo 8), realizando-se a leitura dos valores correspondentes para cada partícula, ou seja, três números reais para cada partícula correspondento a posição; três números reais para cada partícula correspondendo a velocidade inicial; um número real para cada partícula correspondendo a massa; e um caractere alfabético para cada partícula correspondendo ao tipo da partícula. Na linha 43, encerra-se o laço de repetição iniciado no algoritmo 8.

Inicialização da estrutura de dados das partículas (SistemaParticula::_inicializa)

Aloca dinâmicamente os vetores para armazenar: posição, velocidade, massa, força, tipo de cada partícula. Inicializa as estruturas de dados utilizadas no cálculo das velocidades média, e do desvio padrão amostral.

Algoritmo 11 Inicialização da Estrutura de Dados

1: $r \leftarrow$ endereço da região de memória para se armazenar $3 * qt$ números reais ▷ posição da partícula
2: $v \leftarrow$ endereço da região de memória para se armazenar $3 * qt$ números reais ▷ velocidade inicial da partícula
3: $m \leftarrow$ endereço da região de memória para se armazenar qt números reais ▷ massa da partícula
4: $f \leftarrow$ endereço da região de memória para se armazenar $3 * qt$ números reais ▷ força de interação correspondente a partícula
5: $ti \leftarrow$ endereço da região de memória para se armazenar qt números inteiros ▷ tipo da partícula
6:
7: $t \leftarrow 0.0$ ▷ instante de tempo
8: $dt \leftarrow 1.0e-2$ ▷ intervalo de tempo a ser utilizado
9: ▷ Inicializa os vetores que armazenam a velocidade média e o desvio padrão amostral
10: **para** $i \leftarrow 0, 99$ **faça**
11: $vm[i] \leftarrow 0.0$
12: $cv[i] \leftarrow 0$
13: **fim para**
14:
15: **se** não conseguiu alocar memória **então**
16: encerra o programa com mensagem de erro
17: **fim se**

Leitura de dados de 3 componentes (SistemaParticula::import_3f)

Realiza a immportação dos valores numéricos correspondentes a um conjunto de 3 valores. Recebe uma cadeia de caracteres obtida através da leitura de um token do arquivo xml a separa de três em três valores reais atribuindo ao atributo correspondente de cada uma das partículas. Por exemplo velocidade nas direções x, y e z da partícula correspondente. A figura 3.2 ilustra o armazenamento dos dados na memória representada pelo vetor d no algoritmo 12, apresentado a seguir.

Algoritmo 12 Leitura de dados de 3 componentes

1: $s1 \leftarrow$ *linha de caracteres representando os valores lidos do arquivo xml*
2: $x \leftarrow$ extrai os caracteres contidos em $s1$ até o próximo separador e converte para real
3: $y \leftarrow$ extrai os caracteres contidos em $s1$ até o próximo separador e converte para real
4: $z \leftarrow$ extrai os caracteres contidos em $s1$ até o próximo separador e converte para real
5: $d[0] \leftarrow x$
6: $d[1] \leftarrow y$
7: $d[2] \leftarrow z$
▷ percorre todos os grupos de valores
8: **para** $i \leftarrow 1, (qt-1)$ **faça**
▷ extrai a partir dos caracteres os valores correspondentes
9: $x \leftarrow$ extrai os caracteres contidos em $s1$ até o próximo separador e converte para real
10: $y \leftarrow$ extrai os caracteres contidos em $s1$ até o próximo separador e converte para real
11: $z \leftarrow$ extrai os caracteres contidos em $s1$ até o próximo separador e converte para real
▷ armazena os números obtidos nas respectivas posições
12: $d[i*3] \leftarrow x$
13: $d[i*3+1] \leftarrow y$
14: $d[i*3+2] \leftarrow z$
15: **fim para**

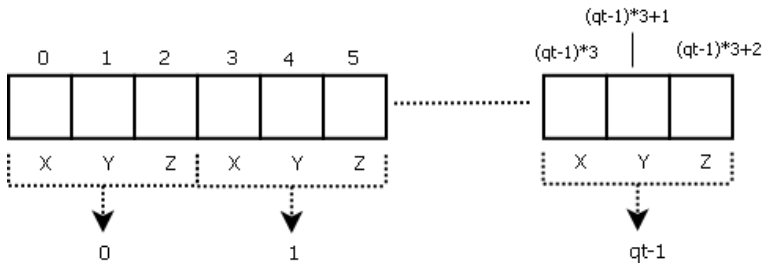

Figura 3.2: Representação do armazenamento dos dados na memória, representada pelo vetor d.

Leitura de dados de 1 componente(SistemaParticula::import_1f)

Realiza a importação do valor numérico correspondente a massa de cada partícula. Recebe uma cadeia de caracteres obtida através da leitura de um token do arquivo xml a separa de um em um valor real atribuindo ao atributo correspondente de cada uma das partículas. Por exemplo massa da partícula correspondente.

Algoritmo 13 Leitura de dados de 1 componente

1: $s1 \leftarrow$ *linha de caracteres representando os valores lidos do arquivo xml*
2: $x \leftarrow$ extrai os caracteres contidos em $s1$ até o próximo separador e converte para real
3: $d[0] \leftarrow x$
4: **para** $i \leftarrow 1, (qt - 1)$ **faça**
5: $x \leftarrow$ extrai os caracteres contidos em $s1$ até o próximo separador e converte para real
6: $d[i] \leftarrow x$
7: **fim para**

Importação dos caracteres relativos ao tipo de cada partícula (SistemaParticula::import_1i)

Recebe uma cadeia de caracteres obtida através da leitura de um token do arquivo xml e a separa de um em um caracter, o qual será convertido para o código inteiro do tipo correspondente, ou seja 'A' tipo 0, 'B' tipo 1 e assim por diante, atribuindo ao atributo correspondente de cada uma das partículas.

Algoritmo 14 Leitura de dados de 1 componente

1: $p \leftarrow 0$
2: $i \leftarrow 0$
3: **Enquanto** ($p <$ quantidade de caracteres) **faça**
4: **se** (caracter $\geqslant' A'$ & caracter $\leqslant' Z'$) **então**
5: $x \leftarrow$ caracter $-' A'$
6: $d[i] \leftarrow x$
7: $i \leftarrow i + 1$
8: **fim se**
9: $p \leftarrow p + 1$
10: **fim Enquanto**

No algoritmo 11 da linha 3 até a linha 11 temos um laço de repetição que irá percorrer toda a

cadeia de caracteres, e encontrando um caractere alfabético maiúsculo, linha 4, utilizará como base a tabela *ASCII* e fará a conversão da letra para o correspondente número inteiro de 0 a 27, conforme apresentado na linha 5. Na linha 6, este número é armazenado e o processo continua no próximo caracter até que todos os caracteres tenham sido convertidos.

3.4 Construção da Lista de Partículas

Constroi a lista de células conectadas. Inicialmente inicializa com -1 o arranjo que armazena o *id* da última partícula da listas de partículas da respectiva célula que a contém (identificado como *head*). Para todas as partículas aplica o conceito de fronteira periódica (PBC), ou seja se uma determinada coordenada da partícula for menor que a menor coordenada do domínio da simulação incrementa esta coordenada da partícula com com o comprimento do domínio de simulação nesta dimensão, caso contrário decrementa esta coordenada da partícula. Em seguida calcula as coordenadas da partícula relativa ao grid de células, calcula o indice da célula correspondente no arranjo de células que representa o grid. Armazena o conteúdo de *head* da célula identificada, no arranjo que representa a lista ligada na posição correspondente a partícula que esta sendo considerada. Armazena em *head* da célula identificada o número da partícula.

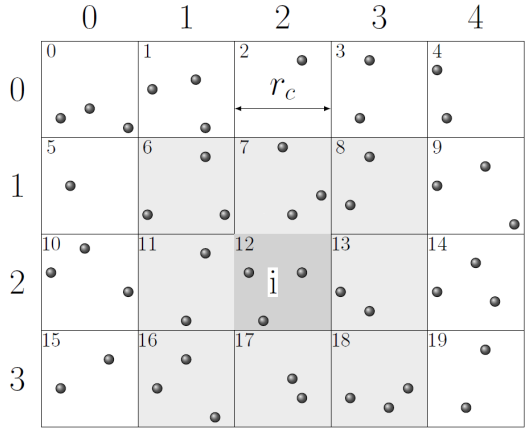

Figura 3.3: Representação 2D da vizinhança para construção da lista de células.
(FRENKEL E SMIT, 1996)

A figura 3.3 mostra para o caso 2D o domínio da simulação e dividido em células quadradas de tamanho $r_c \times r_c$, no qual a partícula i interage com as partículas que estão na mesma célula ou nas células vizinhas (em 2D existem 9 células; e em 3D, 27 células). (FRENKEL E SMIT, 1996)

Algoritmo 15 Construção da lista de células

```
1:  para i ← 0, pnc − 1 faça
2:      head[i] ← −1
3:  fim para
4:  para i ← 0, qt − 1 faça
5:      para k ← 0, 2 faça
6:          se (r[i ∗ 3 + k] < 0) então
7:              r[i ∗ 3 + k] ← r[i ∗ 3 + k] + l[k]
8:          senão se (r[i ∗ 3 + k] > l[k]) então
9:              r[i ∗ 3 + k] ← r[i ∗ 3 + k] − l[k]
10:         fim se
11:         kc[k] ← ⌊r[i ∗ 3 + k] ∗ nc[k]/l[k]⌋
12:     fim para
13:     ind ← index(kc,nc)
14:     ll[i] ← head[ind]
15:     head[ind] ← i
16: fim para
```

Podemos observar que para efeitos de otimização o laço de repetição com variável de controle representado pelas linhas de 5 a 6, pode ser aberto como apresentado a seguir:

Algoritmo 16 Abertura do laço de repetição

1: **se** $(r[i*3] < 0)$ **então**
2: $\quad r[i*3] \leftarrow r[i*3] + l[0]$
3: **senão se** $(r[i*3] > l[0])$ **então**
4: $\quad r[i*3] \leftarrow r[i*3] - l[0]$
5: **fim se**
6: **se** $(r[i*3+1] < 0)$ **então**
7: $\quad r[i*3+1] \leftarrow r[i*3+1] + l[1]$
8: **senão se** $(r[i*3+1] > l[1])$ **então**
9: $\quad r[i*3+1] \leftarrow r[i*3+1] - l[1]$
10: **fim se**
11: **se** $(r[i*3+2] < 0)$ **então**
12: $\quad r[i*3+2] \leftarrow r[i*3+2] + l[2]$
13: **senão se** $(r[i*3+2] > l[2])$ **então**
14: $\quad r[i*3+2] \leftarrow r[i*3+2] - l[2]$
15: **fim se**
16: $kc[0] \leftarrow \lfloor r[i*3+0] * nc[0]/l[0] \rfloor$
17: $kc[1] \leftarrow \lfloor r[i*3+1] * nc[1]/l[1] \rfloor$
18: $kc[2] \leftarrow \lfloor r[i*3+2] * nc[2]/l[2] \rfloor$

3.5 Construção da Lista de Partículas Vizinhas

Inicialmente zera o arranjo de inteiros que armazena a quantidade de vizinhos de cada partícula. Em seguida, para todas as partículas, calcula o índice das coordenadas da partícula em cada uma das dimensões do grid de células, com base nas coordenadas do grid percorre as 26 células vizinhas, aplicando quando necessário a condição de fronteira periódica. Para a célula em questão calcula o índice correspondente da célula no arranjo de células, recuperando o *id* da primeira partícula contida na lista de partículas da célula, observando que as células que não possuem partículas o elemento correspondente no arranjo *head* contém o valor -1. Se o índice da partícula "vizinha" for menor que o da partícula "central" calcula a distância *mínima imagem* entre a partícula "central" e a partícula "vizinha", se a distância for menor ou iqual a $1.5 \times r_c$ inclui o índice da partícula "vizinha" no arranjo bidimensional conforme proposto por (ANDERSON *e outros*, 2008), na linha correspondente ao número de partículas vizinhas da partícula "central" e na coluna correspondente

ao índice da partícula “central”, incrementando o contador de vizinhas da partícula “central”, pega o índice da próxima partícula da célula, refazendo os passos até que o índice seja igual a -1, o que indica chegou o fim da lista de partículas da célula em questão.

A figura 3.4 ilustra o caso tridimensional no qual se tem uma célula e sua vizinhança, a célula em questão contém 13 partículas, e para uma determinada partícula a figura apresenta a região delimitada pelo raio de corte r_c, sendo que esta partícula possui por sua vez 4 partículas vizinhas.

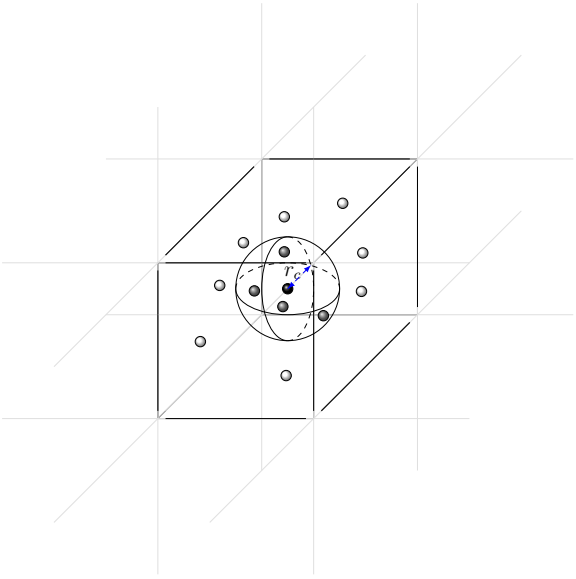

Figura 3.4: Representação das células conectadas e a vizinhança de uma determinada partícula.

Calcula a menor distância entre duas partículas (SistemaParticula::minimum_image)

Considerando inicialmente um domínio de simulação que contém duas partículas, conforme ilustrado na figura 3.5:

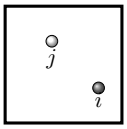

Figura 3.5: Representação das células conectadas e a vizinhança de uma determinada partícula.

A este domínio será aplicado o conceito de fronteira periódica, teremos agora nossa fronteira sendo representada pela figura abaixo, figura 3.6:

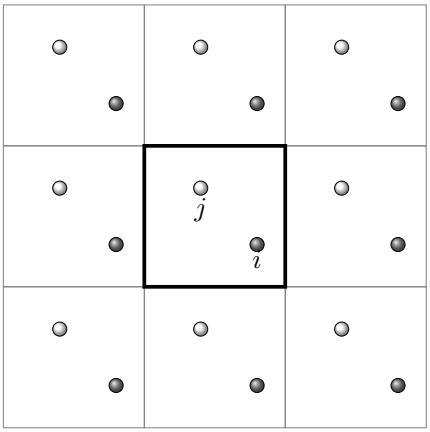

Figura 3.6: Representação das células conectadas e a vizinhança de uma determinada partícula.

Ao se considerar a distância da partícula i com todas as partículas j existentes, considerando as vizinhança de células teremos, a situação como ilustrada na figura 3.7

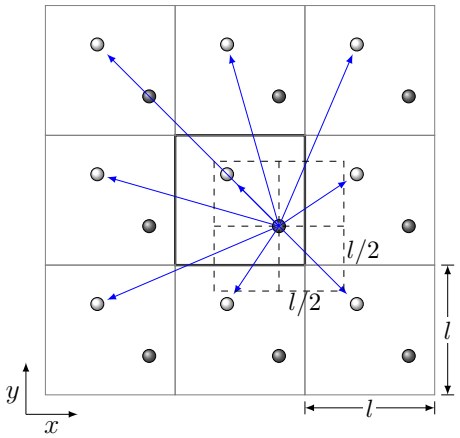

Figura 3.7: Representação das distâncias entre as partículas 'j' e a partícula i.

Na figura 3.7 observamos que as imagens da partícula j estão a distâncias variadas da partícula i, sendo que nos interessa a mais próxima, ou seja, a que esteja a mínima distância da partícula i, para se evitar os gastos computacionais comparando distâncias, podemos observar que algumas destas imagens estão, considerando os eixos, a uma distância menor que $l/2$ e outras a uma distância maior que $l/2$. Na figura 3.8 observamos que para estas partículas que estão a uma distância maior que $l/2$ se subtrairmos ou adicionarmos, dependendo do caso, o valor de l a suas coordenadas x e/ou y, teremos como resultado final a coordenada correspondente a imagem mais próxima , consequentemente a distância calculada será o que se chama de *minimum-image*.

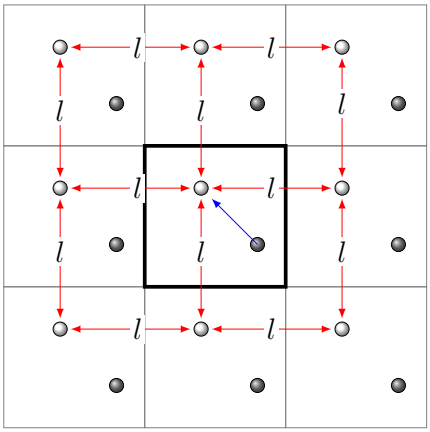

Figura 3.8: Representação das células conectadas e a vizinhança de uma determinada partícula.

Como podemos observar na figura 3.8, o algoritmo 17 verifica se a distância relativa entre as partículas ultrapassa a metade da dimensão da célula, linha 4, neste caso soma ou subtrai a dimensão da célula de maneira a sempre considerar a imagem mais próxima dentre as imagens da partícula em consideração.

Algoritmo 17 Calcula a distância correspondente a imagem mínima

1: $dr \leftarrow 0$
2: **para** $k \leftarrow 0, 2$ **faça**
3: $\quad dx \leftarrow r[i * 3 + k] - r[j * 3 + k]$
4: $\quad$ **se** $(abs(dx) > (l[k] * 0.5))$ **então**
5: $\quad\quad dx \leftarrow dx - sinal(dx) * l[k]$
6: $\quad$ **fim se**
7: $\quad rv[k] \leftarrow dx$
8: $\quad dr \leftarrow dr + dx^2$
9: **fim para**
10: $rv[3] \leftarrow dr$

No algoritmo 18, apresentado a seguir, o laço de repetição expresso na linha 2 do algoritmo 17 e aberto de maneira a evidenciar o cálculo da distância mínima, focando na possibilidade de uma possível otimização do desempenho.

Algoritmo 18 Calcula a distância imagem mínima com a repetição aberta

1: $dx \leftarrow r[i * 3] - r[j * 3]$
2: **se** $(abs(dx) > (l[0] * 0.5))$ **então**
3: $\quad dx \leftarrow dx - sinal(dx) * l[d]$
4: **fim se**
5: $rv[0] \leftarrow dx$
6: $dr \leftarrow dr + dx^2$
7: $dx \leftarrow r[i * 3 + 1] - r[j * 3 + 1]$
8: **se** $(abs(dx) > (l[1] * 0.5))$ **então**
9: $\quad dx \leftarrow dx - sinal(dx) * l[d]$
10: **fim se**
11: $rv[1] \leftarrow dx$
12: $dr \leftarrow dr + dx^2$
13: $dx \leftarrow r[i * 3 + 2] - r[j * 3 + 2]$
14: **se** $(abs(dx) > (l[2] * 0.5))$ **então**
15: $\quad dx \leftarrow dx - sinal(dx) * l[d]$
16: **fim se**
17: $rv[2] \leftarrow dx$
18: $dr \leftarrow dr + dx^2$
19: $rv[3] \leftarrow dr$

Com base no cálculo da imagem mínima expresso pelo algoritmo 17, podemos construir a lista de partículas vizinhas de uma determinada partícula como apresentado no algoritmo 19, apresentado a seguir.

Algoritmo 19 Construção da lista de partículas vizinhas

1: **para** $i \leftarrow 0, qt-1$ **faça**
2: $NN[i] \leftarrow 0$
3: **fim para**
4: **para** $i \leftarrow 0, qt-1$ **faça**
5: $ic[0] \leftarrow \lfloor r[i*3] * nc[0]/l[0] \rfloor$
6: $ic[1] \leftarrow \lfloor r[i*3+1] * nc[1]/l[1] \rfloor$
7: $ic[2] \leftarrow \lfloor r[i*3+2] * nc[2]/l[2] \rfloor$
8: **para** $kc[0] \leftarrow ic[0]-1, ic[0]$ **faça**
9: **para** $kc[1] \leftarrow ic[1]-1, ic[1]$ **faça**
10: **para** $kc[2] \leftarrow ic[2]-1, ic[2]$ **faça**
11: Tratamento das condições de fronteira periódica
12: $ind \leftarrow index(kcx,nc)$
13: $k \leftarrow head[ind]$
14: Construção da vizinhança da partícula i
15: **fim para**
16: **fim para**
17: **fim para**
18: **fim para**

Na construção da lista de partículas vizinhas, o algorimo 19, na linha 11, considera a questão da periodicidade do domínio de simulação, a qual é apresentada em detalhes no algorimo 20.

Algoritmo 20 Construção da lista de partículas vizinhas, periodicidade.

1: **para** $l \leftarrow 0, 2$ **faça**
2: **se** $(kc[l] < 0)$ **então**
3: $kcx[l] \leftarrow nc[l] + kc[l]$
4: **senão se** $kc[l] \leq nc[l]$ **então**
5: $kcx[l] \leftarrow kc[l]/nc[l]$
6: **senão**
7: $kcx[l] \leftarrow kc[l]$
8: **fim se**
9: **fim para**

Considerada a periodicidade das fronteiras, para uma determinada partícula são consideradas as partículas contídas na célula em questão e nas suas vizinhas, vide algoritmo 21.

Algoritmo 21 Construção da lista de partículas vizinhas, vizinhança

1: **Enquanto** $k \geq 0$ **faça**
2: $dr \leftarrow 0$
3: $dr \leftarrow minimum\_image(i, k, rd)$
4: **se** $i \neq k$ **então**
5: **se** $dr \leq (rc * 1.5)^2$ **então**
6: **se** $i < k$ **então**
7: $NBL[NN[i]][i]] \leftarrow k$
8: $NN[i] \leftarrow NN[i] + 1$
9: **fim se**
10: **fim se**
11: **fim se**
12: $k \leftarrow ll[k]$
13: **fim Enquanto**

3.6 Verlet 1

Para todas as partículas, para quais não forem partículas fixas que interagem mas não se deslocam, calcula a aceleração, através da aceleração calcula o deslocamento e a velocidade da partícula:

$$c_0 = 1.0 \tag{3.2}$$

$$c_1 = 1.0 \tag{3.3}$$

$$c_2 = 0.5 \tag{3.4}$$

$$a_i[d] = d_i[d]/m_i \tag{3.5}$$

$$d_i[d] = c_1 \times v_i[d] \times \delta t + c_2 \times a_i[d] \times \delta t^2 \tag{3.6}$$

$$v_i[d] = c_0 \times v_i[d] + (c_1 - c_2) \times a_i[d] \times \delta t \tag{3.7}$$

Na sequência verifica se ocorreu a reflexão na fronteira, no nosso caso utilizamos a reflexão do tipo *bouncy-forward*.

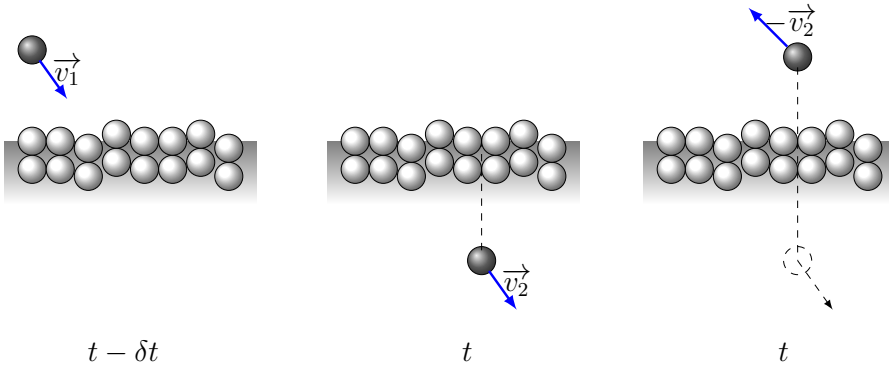

Figura 3.9: Representação da aplicação da reflexão para frente em uma partícula.

Algoritmo 22 Integração Velocity-Verlet primeira etapa

1: **para** $i \leftarrow 0, qt - 1$ **faça**
2: **se** partícula i não for fixa **então**
3: **para** $k \leftarrow 0, 2$ **faça**
4: $a \leftarrow f[i * 3 + k]/m[i]$
5: $ds \leftarrow v[i * 3 + k] * dt + x * dt^2/2$
6: $dv[k] \leftarrow v[i * 3 + k] + a * dt/2$
7: $r[i * 3 + k] \leftarrow r[i * 3 + k] + ds$
8: $v[i * 3 + k] \leftarrow v[i * 3 + k] + dv[k]$
9: **fim para**
10: **se** $r[i * 3 + 2] < 3.0$ **então**
11: $dr \leftarrow 3.0 - r[i * 3 + 2]$
12: $r[i * 3 + 2] + \leftarrow 2 * dr$
13: $v[i * 3] \leftarrow -dv[0]$
14: $v[i * 3 + 1] \leftarrow -dv[1]$
15: $v[i * 3 + 2] \leftarrow -dv[2]$
16: **senão se** $r[i * 3 + 2] > (l[2] - 3.0)$ **então**
17: $dr \leftarrow r[i * 3 + 2] - (l[2] - 3.0)$
18: $r[i * 3 + 2] - \leftarrow 2 * dr$
19: $v[i * 3] \leftarrow -dv[0]$
20: $v[i * 3 + 1] \leftarrow -dv[1]$
21: $v[i * 3 + 2] \leftarrow -dv[2]$
22: **fim se**
23: **fim se**
24: **fim para**

3.7 Forças

Para todas as partículas inicializa o arranjo de forças com a força externa. Em seguida para cada uma das partículas, percorre a lista de partículas vizinhas e para as que estão a uma distância menor que r_c realiza o cálculo das forças.

Para o cálculo da força conservativa, calcula a magnitude do vetor distância, identifica qual

coeficiente de interação será utilizado, calcula o fator de ponderação e a força conservativa correspondente.

Algoritmo 23 Cálculo da Força Conservativa

1: $dr \leftarrow minimum\_image(i,k,rd)$
2: **se** $dr \leq r_c^2$ **então**
3: $\quad mag \leftarrow \mid rd \mid$
4: $\quad w \leftarrow (1 - mag/r_c)$
5: $\quad aux \leftarrow 1/mag$
6: $\quad fc \leftarrow (\alpha_{ij} \times w)$
7: **fim se**

Para o cálculo da força dissipativa, calcula o vetor unitário correspondente ao vetor distância, a velocidade relativa entre as partículas, a componente na direção do vetor unitário calulado e o coeficiente de fricção γ na sequência calcula a força dissipativa correspondente.

Algoritmo 24 Cálculo a Força Dissipativa

1: $dr \leftarrow minimum\_image(i,k,rd)$
2: **se** $dr \leq r_c^2$ **então**
3: $\quad mag \leftarrow \mid rd \mid$
4: $\quad w \leftarrow (1 - mag/r_c)$
5: $\quad aux \leftarrow 1/mag$
6: $\quad \overrightarrow{e} \leftarrow aux \times \overrightarrow{rd}$
7: $\quad \overrightarrow{v} \leftarrow \overrightarrow{v}_i - \overrightarrow{v}_k$
8: $\quad dot \leftarrow \overrightarrow{e} \cdot \overrightarrow{v}$
9: $\quad fd \leftarrow ((-\gamma_{ij} \times w^2) \times dot)$
10: **fim se**

Para o cálculo da força aleatória, utiliza um gerador de ruído branco gaussianos com média zero e variância um, com o peso w e a amplitude do ruído dado por σ calcula a força aleatória correspondente.

Algoritmo 25 Cálculo da Força Aleatória

$dr \leftarrow minimum\_image(i,k,rd)$
se $dr \leq r_c^2$ **então**
    $mag \leftarrow \mid rd \mid$
    $w \leftarrow (1 - mag/r_c)$
    $fr \leftarrow \sigma_{ij} \times w \times gaussian(0,1)/\sqrt{\delta t}$
fim se

A seguir apresentamos o algoritmo completo da junção do cálculo das três componentes.

Algoritmo 26 Cálculo das Forças

$dr \leftarrow minimum\_image(i,k,rd)$
se $dr \leq r_c^2$ **então**
    $mag \leftarrow \mid rd \mid$
    $aux \leftarrow 1/mag$
    $\overrightarrow{e} \leftarrow aux \times \overrightarrow{rd}$
    $w \leftarrow (1 - mag/r_c)$
    $fc \leftarrow (\alpha_{ij} \times w)$ ▷ Calcula Força Conservativa
    $\overrightarrow{v} \leftarrow \overrightarrow{v}_i - \overrightarrow{v}_k$
    $dot \leftarrow \overrightarrow{e} \cdot \overrightarrow{v}$
    $fd \leftarrow ((-\gamma_{ij} \times w^2) \times dot)$ ▷ Calcula Força Dissipativa
    $fr \leftarrow \sigma_{ij} \times w \times gaussian(0,1)/\sqrt{\delta t}$ ▷ Calcula Força Aleatória
    $\overrightarrow{f} \leftarrow (fc + fd + fr) \times \overrightarrow{e}$

    se partícula i não for fixa **então**
        $\overrightarrow{f}_i \leftarrow \overrightarrow{f}_i + \overrightarrow{f}$
    fim se

    ▷ Aplica a terceira lei de Newton
    se partícula k não for fixa **então**
        $\overrightarrow{f}_k \leftarrow \overrightarrow{f}_k - \overrightarrow{f}$
    fim se
fim se

3.8 Verlet 2

A integração pelo método Velocity-Verlet em sua última etapa atualiza velocidade das partículas, vide figura 2.15, com base na velocidade anterior e aceleração correspondente. O algoritmo 28 apresentado a seguir, tem a finalidade de atualizar a velocidade. Observamos que está atualização é realizada somente para aquelas partículas que não são fixas.

Algoritmo 27 Integração Velocity-Verlet segunda etapa

1: **para** $i = 0$,$qt - 1$ **faça**
2: **se** partícula i não for fixa **então**
3: $v[i * 3] \leftarrow v[i * 3] + f[i * 3] * dt/(2 * m[i])$
4: $v[i * 3 + 1] \leftarrow v[i * 3 + 1] + f[i * 3 + 1] * dt/(2 * m[i])$
5: $v[i * 3 + 2] \leftarrow v[i * 3 + 2] + f[i * 3 + 2] * dt/(2 * m[i])$
6: **fim se**
7: **fim para**

Uma simplificação neste algoritmo, seria nas linhas de 3 a 5, considerar a massa como unitária e pré-calcular $dt/2$, eliminando assim uma operação de multiplicação e uma operação de divisão para cada dimensão.

Algoritmo 28 Integração Velocity-Verlet segunda etapa

1: $dt2 \leftarrow 0.5 * dt$
2: **para** $i = 0$,$qt - 1$ **faça**
3: **se** partícula i não for fixa **então**
4: $v[i * 3] \leftarrow v[i * 3] + f[i * 3] * dt2$
5: $v[i * 3 + 1] \leftarrow v[i * 3 + 1] + f[i * 3 + 1] * dt2$
6: $v[i * 3 + 2] \leftarrow v[i * 3 + 2] + f[i * 3 + 2] * dt2$
7: **fim se**
8: **fim para**

4 Resultados e Discussão

4.1 Validação das Equações de movimento

Implementou-se em uma planilha eletrônica o modelo usado no método DPD, de modo a verificar se o motor de simulação estava corretamente implementado.

Para verificar a contribuição de cada uma das forças, foi construída uma planilha, na qual para uma dada disposição de três partículas, e considerando apenas o movimento em uma direção, foram calculadas as forças dissipativas. A figura 4.1 ilustra a disposição das partículas e seus atributos.

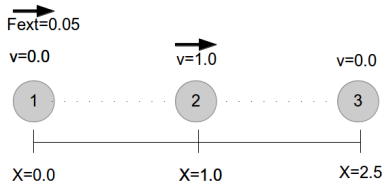

Figura 4.1: Disposição das partículas

A tabela 4.1 apresenta os elementos contidos na planilha: a identificação das colunas, o valor inicial e as fórmulas utilizadas. As fórmulas utilizadas são correspondentes as equações para o cálculo da força dissipativa do método DPD, vide equação 2.4.

Coluna	Descrição	Valor Inicial	Fórmula
A	instante de tempo	0,00	$= A_{t-1} + dt/2$
B	posição de P_1	0,00	0,00
C	velocidade de P_2	1,00	$= P_{t-1}$
D	posição de P_2	1,0	$= Q_{t-1}$
E	posição de P_3	2,5	2,5
F	módulo do valor absoluto da posição relativa P_2 e P_1	1,0	$= ABS(D_t - B_t)$
G	módulo do valor absoluto da posição relativa P_2 e P_3	1,0	$= ABS(D_t - E_t)$
H	velocidade relativa P_2 e P_1	1,0	$= C_t$
I	velocidade relativa P_2 e P_3	1,0	$= C_t$
J	força dissipativa P_2 e P_1	0,0	$= SE(F_t <= rcut; -g * (1 - F_t/rcut)^2 * H_t; 0)$
K	força dissipativa P_2 e P_3	0,0	$= SE(G_t <= rcut; -g * (1 - G_t/rcut)^2 * I_t; 0)$
L	força resultante	$= J_t + K_t$	$= J_t + K_t$
M	velocidade intermediária de P_2	1,0	$= C_t + dt * 0{,}5 * L_t/m$
N	aceleração de P_2	$= L_t/m$	$= L_t/m$
O	deslocamento de P_2	$= C_t * dt + 0{,}5 * N_t * dt * dt$	$= M_t * dt + 0{,}5 * N_t * dt * dt$
P	nova velocidade de P_2	$= C_t + 0{,}5 * N_t * dt$	$= M_t + 0{,}5 * N_t * dt$
Q	nova posição de P_2	$= D_t + O_t$	$= D_t + O_t$

Tabela 4.1: Componentes da planilha para o cálculo da Força Dissipativa

Nesta simulação através da planilha foram utilizados os seguintes parâmetros:

Parâmetros	Valor
rcut	1,0
alfa	25,0
sigma	5,0
gama	12,5
deltat	0,01
massa	1,0

Tabela 4.2: Parâmetros utilizados

Esses parâmetros foram obtidos a partir das curvas apresentadas nas figuras 2.10, 2.11 e 2.12, que relacionam os coeficientes da simulação aos parâmetros físicos de viscosidade de um flúido.

Observamos na figura 4.2 que, considerando apenas a componente dissipativa da força entre as partículas, a partícula P_2 movimenta-se em direção à partícula P_3. A medida que P_2 se aproxima a força de interação entre as partículas P_2 e P_3 aumenta, desacelerando a partícula P_2 até que sua velocidade seja muito pequena. Com a diminuição da velocidade da partícula P_2 a força dissipativa entre as partículas P_2 e P_3 cada vez mais se aproxima de zero. Ressaltamos que em função da partícula P_2 estar além do raio de corte em relação a partícula P_1, a força dissipativa entre elas é nula. Pode-se notar que os resultados obtidos com a planilha são idênticos aos obtidos com o motor de simulação implementado, como podemos observar nas figuras 4.2 e 4.3.

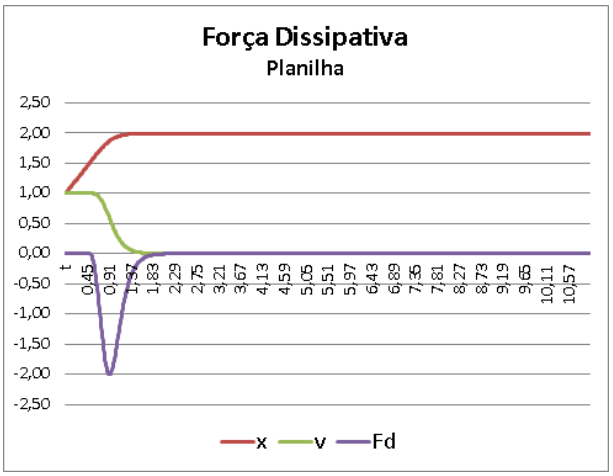

Figura 4.2: Força Dissipativa, Posição e Velocidade obtidos com a planilha

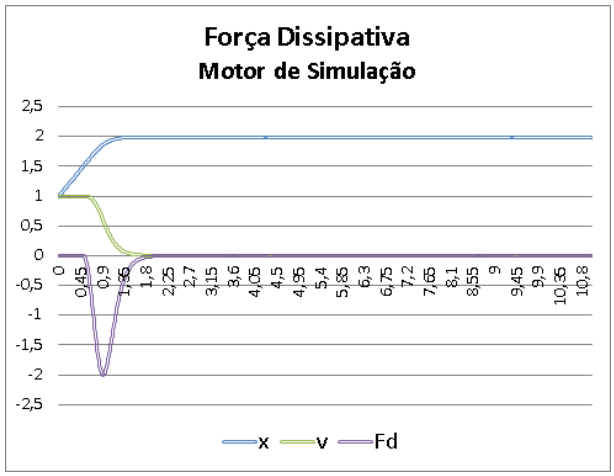

Figura 4.3: Força Dissipativa, Posição e Velocidade obtidos com o motor de simulação

A tabela 4.3 apresenta os elementos contidos na planilha: a identificação das colunas, o valor inicial e as fórmulas utilizadas. As fórmulas utilizadas são correspondentes as equações para o cálculo da força conservativa do método DPD, vide equação 2.3.

Coluna	Descrição	Valor Inicial	Fórmula
A	instante de tempo	0,00	$= A_{t-1} + dt/2$
B	posição de P_1	0,00	0,00
C	velocidade de P_2	1,00	$= P_{t-1}$
D	posição de P_2	1,0	$= Q_{t-1}$
E	posição de P_3	2,5	2,5
F	módulo do valor absoluto da posição relativa P_2 e P_1	1,0	$= ABS(D_t - B_t)$
G	módulo do valor absoluto da posição relativa P_2 e P_3	1,0	$= ABS(D_t - E_t)$
H	velocidade relativa P_2 e P_1	1,0	$= C_t$
I	velocidade relativa P_2 e P_3	1,0	$= C_t$
J	força conservativa P_2 e P_1	0,0	$= SE(F_t <= rcut; alfa * (1 - F_t/rcut) * (D_{t-1} - B_{t-1})/F_t; 0)$
K	força conservativa P_2 e P_3	0,0	$= SE(G_t <= rcut; alfa * (1 - G_t/rcut) * (D_{t-1} - E_{t-1})/G_t; 0)$
L	força resultante	$= J_t + K_t$	$= J_t + K_t$
M	velocidade intermediária de P_2	1,0	$= C_t + dt * 0{,}5 * L_t/m$
N	aceleração de P_2	$= L_t/m$	$= L_t/m$
O	deslocamento de P_2	$= C_t * dt + 0{,}5 * N_t * dt * dt$	$= M_t * dt + 0{,}5 * N_t * dt * dt$
P	nova velocidade de P_2	$= C_t + 0{,}5 * N_t * dt$	$= M_t + 0{,}5 * N_t * dt$
Q	nova posição de P_2	$= D_t + O_t$	$= D_t + O_t$

Tabela 4.3: Componentes da planilha para o cálculo da Força Conservativa

Observamos no gráfico 4.4 que, considerando apenas a componente conservativa da força, a partícula P_2 movimenta-se em direção a partícula P_3, a medida que se aproxima, a força de interação entre as partículas P_2 e P_3 aumenta desacelerando a partícula P_2, até que ela inverte seu movimento, neste instante a magnitude da força conservativa gerada pela interação entre as partículas P_2 e P_3 é máxima, com o afastamento da partícula P_2 a força conservativa entre as partículas diminui, a partir de um ponto a particula P_2 se afasta da partícula P_3 ficando além do raio de corte, momento no qual a força conservativa passa a ser zero, posteriormente quando a partícula P_2 fica a uma distância menor do que o raio de corte da partícula P_1 a força conservativa passa a atuar novamente na partícula P_2, fazendo com que sua velocidade em módulo diminua até o ponto em que ela inverte seu movimento, ponto no qual a intensidade da força conservativa é máxima. Notamos também que os resultados obtidos com a planilha são idênticos aos obtidos com o motor de simulação implementado, como podemos observar nas figuras 4.4 e 4.5.

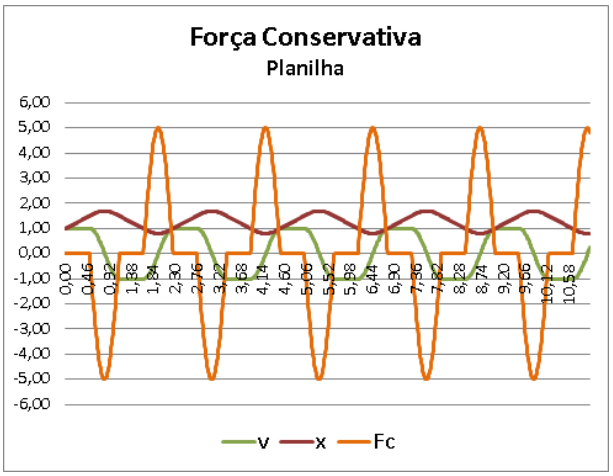

Figura 4.4: Força Conservativa, Posição e Velocidade obtidos com a planilha

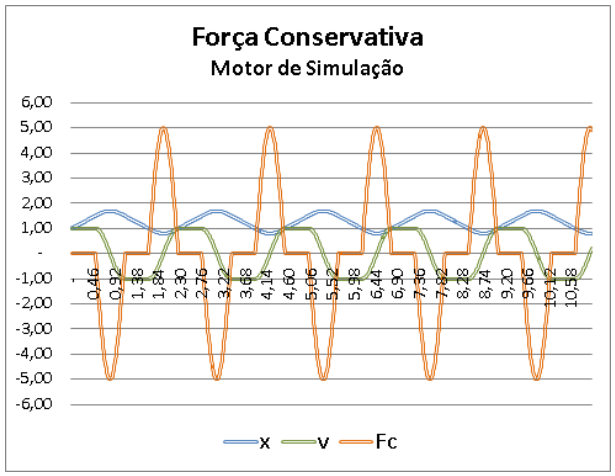

Figura 4.5: Força Conservativa, Posição e Velocidade obtidos com o motor de simulação

No gráfico 4.6 são apresentadas as Forças Conservativa e Dissipativa atuando na interação entre as três partículas do experimento, observamos que a força conservativa possui uma magnitude maior que a força dissipativa, modulando assim o movimento da partícula, é possível também perceber pelo gráfico que as magnitudes das forças diminuem com o tempo.

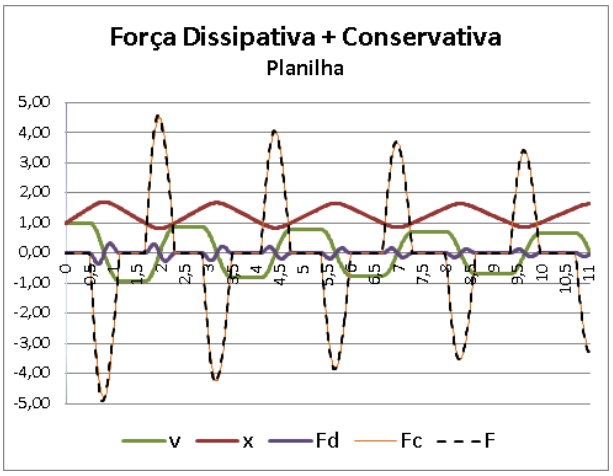

Figura 4.6: Força Dissipativa + Conservativa, Posição, Velocidade obtidos com a planilha

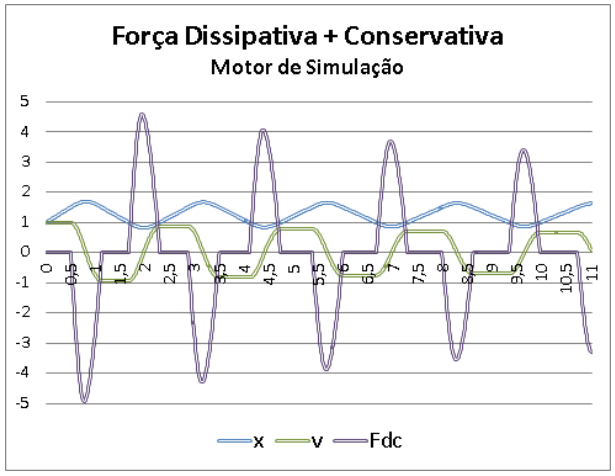

Figura 4.7: Força Dissipativa + Conservativa, Posição, Velocidade obtidos com o motor de simulação

Os resultados obtidos com a planilha são idênticos aos obtidos com o motor de simulação implementado, como podemos observar nas figuras 4.6 e 4.7. No gráfico 4.7 podemos observar

ainda, que a amplitude da velocidade diminui e o tempo que a partícula leva para ir de um extremo ao outro aumenta.

Com o apresentado nesta seção, observamos que a interação entre duas partículas, considerando apenas as forças dissipativa e conservativa não conduz a uma situação em que a velocidade máxima absoluta aumente indefinidamente, tendendo a velocidade máxima da partícula diminuir com o passar das iterações. Comprovando assim, no que diz respeito as forças conservativas e dissipativas, que as equações implementadas estão coerentes com o proposto por Español e Warren (1995) em seu trabalho.

Na próxima seção serão apresentados alguns resultados da geração de números pseudo-aleatórios, que através dos quais será calculada a componente aleatória da força.

4.2 Validação da geração de números pseudo-aleatórios

O método DPD permite construir um modelo de partículas estocástico para sistemas fluídicos (HOOGERBRUGGE E KOELMAN, 1992). Hoogerbrugge e Koelman (1992) utilizaram uma amostragem de números aleatórios a partir de uma distribuição uniforme em simulações 2D. Neste sentido é interessante verificar se o gerador de números pseudo-aleatórios utilizado é adequado, evitando-se assim resultados não condizentes com o método. De maneira que esta secção tem como objetivo apresentar alguns geradores, certificando de sua corretude e analisando seu desempenho, não pretendendo ser uma discussão aprofundada do assunto por este não ser o foco do trabalho.

Na figura 4.8 é apresentado o histograma obtido com base em um conjunto 10.000 números pseudo-aleatórios gerados a partir da distribuição uniforme [0,1] utilizada. A qualidade deste gerador pode ser avaliada através do teste de hipótese de *Kolmogorov-Smirnov* com probabilidade 0,88596, confirmando que a amostra provem de uma distribuição uniforme [0,1].

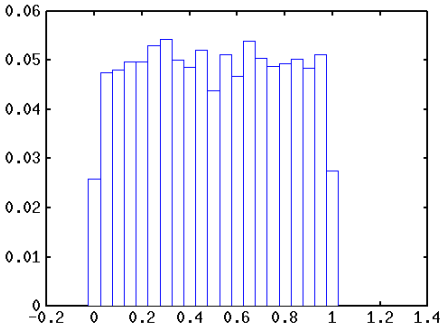

Figura 4.8: Histograma da amostra de números pseudo-aleatórios obtido com o gerador utilizado

A força estocástica é responsável pelo movimento Browniano das partículas do polímero ou colóide que é simulado com o DPD, a qual corresponde aos graus de liberdade que foram eliminados no processo de descrição através de uma granulação grossa (ESPAÑOL, 2005). Español (2005) consideram o termo ξ_{ij} na equação 2.5 como sendo um ruído gaussiano. Existem vários geradores de ruído gaussiano, dentre eles consideramos:

- Método do Teorema do Limite Central apresentado por Donadio (2012); A figura 4.9 apresenta o histograma obtido com base em uma amostra de 10.000 números pseudo-aleatórios obtidos com o Método do Teorema do Limite Central. Podemos observar que os valores se concentram em torno de zero e que o histograma tem a forma de uma curva gaussiana. Alguns aspectos estatísticos desta amostra serão avaliados após serem apresentados os demais métodos.

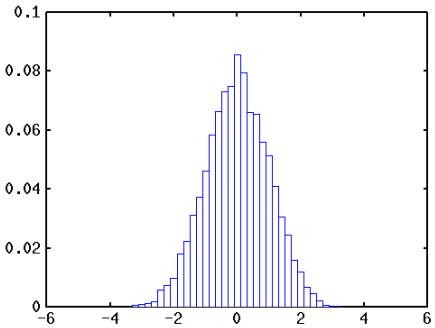

Figura 4.9: Histograma da amostra de números pseudo-aleatórios obtido com o gerador CLTM (algoritmo 1).

- Método Polar de Marsaglia A figura 4.10 apresenta o histograma obtido com base em uma amostra de 10.000 números pseudo-aleatórios obtidos com o Método Polar de Marsaglia. Podemos observar que os valores se concentram em torno de zero e que o histograma tem a forma de uma curva gaussiana. Alguns aspectos estatísticos desta amostra serão avaliados posteriormente.

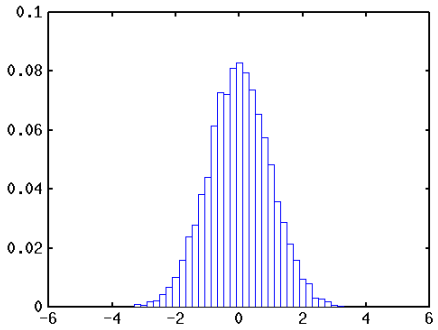

Figura 4.10: Histograma da amostra de números pseudo-aleatórios obtido com o gerador MPM (algoritmo 2).

- Método Ziggurat A figura 4.11 apresenta o histograma obtido com base em uma amostra de 10.000 números pseudo-aleatórios obtidos através do Método Ziggurat. Observamos que os valores se concentram em torno de zero e que o histograma tem a forma de uma curva gaussiana.

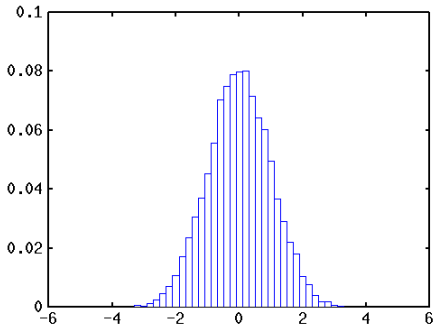

Figura 4.11: Histograma da amostra de números pseudo-aleatórios obtido com o gerador Ziggurat (algoritmo 3).

Pelos histogramas apresentados nas figuras 4.9, 4.10 e 4.11 observamos que os três métodos geram um conjuntos de números pseudo-aleatórios cujos histogramas se aproximam de uma curva

gaussiana com média zero, a fim de determinar qual método será mais adequado consideramos alguns aspectos, o quais são: o tempo necessário para geração do conjunto de números, o achatamento da curva em relação a normal (*kurtosis*), a média, o desvio padrão e o teste de Kolmogorv-Smirnov. Os resultados obtidos são apresentados na tabela 4.12.

Método	**Tempo (ms)**	**Kurtosis**	**Média**	**Desvio Padrão**	**Kolmogorov-Smirnov**
CLTM	16.0	-0.0730	0.0015	1.0021	0.5687
MPM	1.0	0.0796	-0.0103	0.9894	0.2202
Ziggurat	< 1.0	-0.0076	-0.0025	0.9967	0.8866

Figura 4.12: Tempo de geração e estatísticas das amostras. As estatísticas foram obtidas com o pacote Octave.

Podemos observar pelos valores apresentados na tabela 4.12 que o Método Ziggurat apresentou o melhor desempenho computacional e adequação a curva normal, de maneira que ele foi o método escolhido na geração dos números pseudo-aleatórios com distribuição normal. Os experimentos realizados e relatados nestas primeiras seções permitiram um melhor entendimento do equacionamento da interação entre as partículas, na próxima seção serão apresentados alguns resultados das simulações realizadas.

4.3 Perfil de Velocidade

O comportamento hidrodinâmico do fluído DPD pode ser verificado através da comparação da solução analítica com o perfil de velocidade medido, a geometria de referência utilizada foi a de um canal, ilustrada na figura 4.13, na qual o perfil de velocidade pode ser descrito pela equação de Hagen-Poiseuille 4.1. (STEINER, 2009)

$$\nu_x(z) = \frac{1}{2\eta}\left(-\frac{\Delta P}{\Delta x}\right)\left[(\frac{w}{2})^2 - z^2\right] \tag{4.1}$$

com o canal de largura w e comprimento l, gradiente de pressão $-(\Delta P/\Delta x)$ e viscosidade η, determinada pelas condições de fronteira de Lees-Edwards.

Uma maneira de se aferir se um simulador cumpre seu objetivo é utlizá-lo na simulação de alguns problemas clássicos, como por exemplo o fluxo de Poiseuille, uma outra maneira de se validar o simulador é comparar os resultados obtidos com um outro simulador cujos resultados já tenham sido validados, no nosso caso os resultados foram comparados com os resultados teóricos e com o simulador Hoomd-Blue. Primeiramente optamos por confrontar os resultados gerados pelo motor de simulação implementado e pelo Hood-Blue com os resultados analíticos correspondentes. A figura 4.13 apresenta o diagrama esquemático do canal utilizado na simulação do fluxo de um fluído. Observando que nas direções x e y são aplicadas as condições de fronteira periódica.

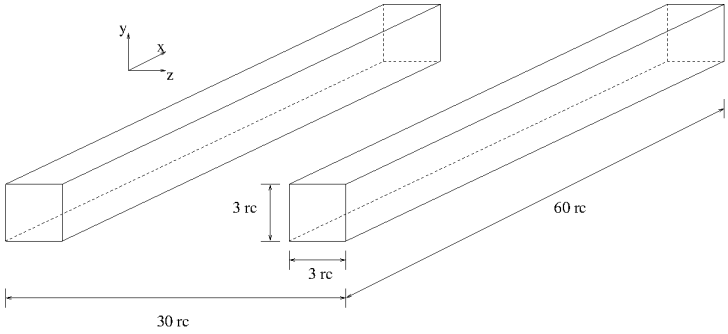

Figura 4.13: Diagrama esquemático da fenda utilizada na simulação

Nesta simulação foram utilizados os parâmetros apresentados na tabela 4.4

Parâmetro	Valor
delta_t	0.02
t_end	220
l	60, 3, 30
nc	40, 2, 20
r_cut	1,0
gamma	4,5
alfa	18.75
alfa1	8.66
sigma	3
f_cte	0.01, 0.0, 0.0

Tabela 4.4: Parâmetros utilizados

As paredes da fenda são formadas por partículas fixas que interagem com as partículas do fluído. A disposição das partículas foi obtida através de uma etapa de “aquecimento” na qual as partículas interagiram durante 100 interações, o que permite uma disposição das partículas da parede, que auxilia na condição de não deslizamento entre o fluído e as paredes do canal.

A figura 4.14 representa um esquema de simulação de um canal no qual a disposição das partículas foi obtida através da etapa de “aquecimento”.

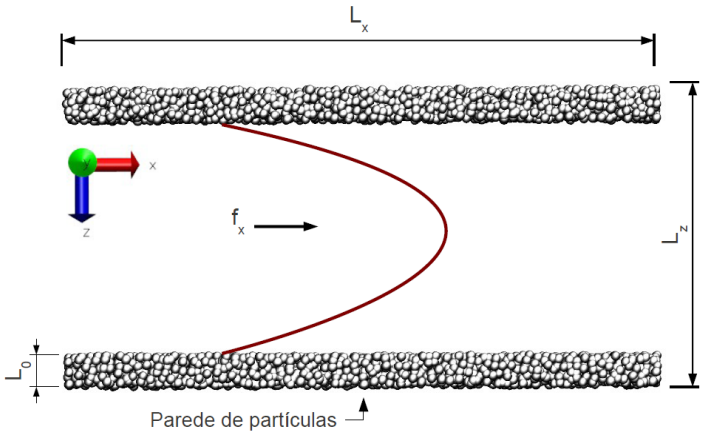

Figura 4.14: Esquema da simulação de um canal

Foi gerado um arquivo de inicialização utilizado tanto nas simulações com o Hoomd-Blue como com o motor de simulação implementado.

No Hoomd-Blue todas as fronteiras são consideradas periódicas, não sendo definido nenhum tipo de reflexão na fronteira entre as paredes do canal e as do fluído. Em função de não haver reflexão, da densidade de partículas na parede e o tipo de interação utilizada, algumas partículas adentram nas paredes, gerando o perfil apresentado na figura 4.15. Podemos observar que nas extremidades a velocidade é proxima de zero e irregular.

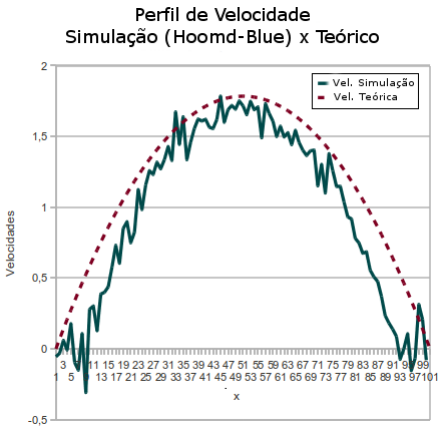

Figura 4.15: Perfil de Velocidade Simulação Hoomd-Blue

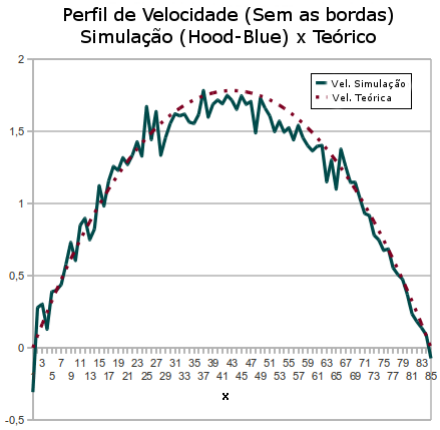

Figura 4.16: Perfil de Velocidade Simulação Hoomd-Blue desconsiderando as bordas

Desconsiderando as seções correspondentes as paredes do canal, obtemos uma melhor representação do perfil de velocidade, o qual é apresentado na figura 4.16. O perfil de velocidade foi obtido considerando apenas a iteração 11.000. Não se tratando de uma média entre um conjunto de iterações a curva obtida do perfil não é suave, contudo podemos observar que ela segue o perfil parabólico do resultado teórico.

A figura 4.17 apresenta o perfil de velocidade obtido pelo motor de simulação implementado. Foi considerada a média das velocidades médias em cada uma das seções do canal nas últimas 100 iterações, tomada como referência a iteração 11.000.

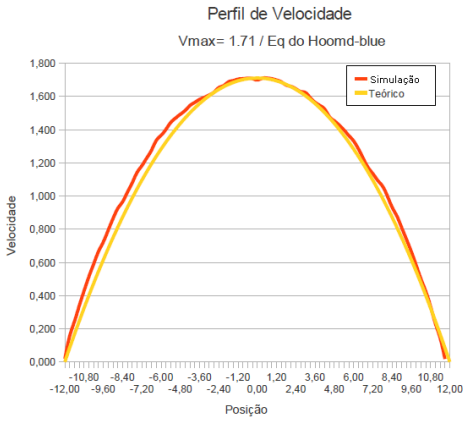

Figura 4.17: Perfil de Velocidade obtido com o motor de simulação implementado

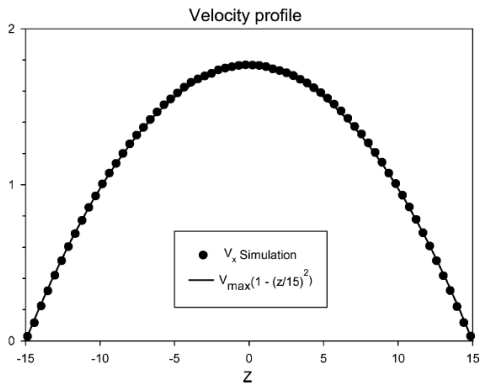

Figura 4.18: Perfil de Velocidade. (DUONG-HONG *e outros*, 2004)

Observamos que os resultados obtidos através do motor de simulação são compatíveis com o simulador Hoomd-Blue e a simulação apresentada no trabalho desenvolvido por Duong-Hong *e outros* (2004).

Na simulação que realizamos utilizando o motor de simulação implementado, para efeito de comparação entre os resultados do motor de simulação e o Hoomd-Blue, foi utilizado o equacionamento para o método DPD utilizado no Hoomd-blue. Neste equacionamento a componente aleatória da força utiliza uma variável aleatória uniformemente distribuída no intervalo $[-1,1]$ (PHILLIPS *e outros*, 2011). O equacionamento uitlizado é apresentado a seguir:

$$F = F_{\mathrm{C}}(r) + F_{\mathrm{R,ij}}(r_{ij}) + F_{\mathrm{D,ij}}(v_{ij})$$

$$\begin{aligned} F_{\mathrm{C}}(r) = &\quad A \cdot w(r_{ij}) \\ F_{\mathrm{R,ij}}(r_{ij}) = &\quad -\theta_{ij}\sqrt{3}\sqrt{\tfrac{2k_b\gamma T}{\Delta t}} \cdot w(r_{ij}) \\ F_{\mathrm{D,ij}}(r_{ij}) = &\quad -\gamma w^2(r_{ij})\,(\hat{r}_{ij} \circ v_{ij}) \end{aligned}$$

$$\begin{aligned} w(r_{ij}) = &\quad (1 - r/r_{\mathrm{cut}}) \quad r < r_{\mathrm{cut}} \\ = &\quad 0 \qquad\qquad r \geq r_{\mathrm{cut}} \end{aligned}$$

4.4 Distribuição de Densidade das Partículas e Temperatura do Sistema

Como discutido nos trabalhos de (LI *e outros*, 2014), (MEHBOUDI E SAIDI, 2013), (FEDOSOV *e outros*, 2008), (DUONG-HONG *e outros*, 2004), em função do método utilizado para impor a condição de não deslizamento na fronteira, observa-se nas proximidades da fronteira a ocorrência de distorções na densidade. Essas distorções ocorrem em maior ou menor grau em função do método escolhido.

No experimento em questão foram utlizadas 17.860 partículas distribuidas em um domínio de simulação de $60r_c \times 3r_c \times 30r_c$, correspondendo a uma densidade de aproximadamente $3{,}3$

$$\frac{17860}{60 * 3 * 30} = \frac{17860}{5400} = 3.3$$

Na figura 4.19 são apresentadas a densidade e temperadura numa seção transversal do canal.

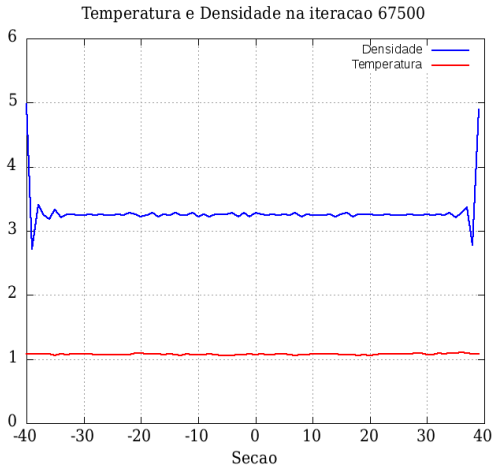

Figura 4.19: Densidade

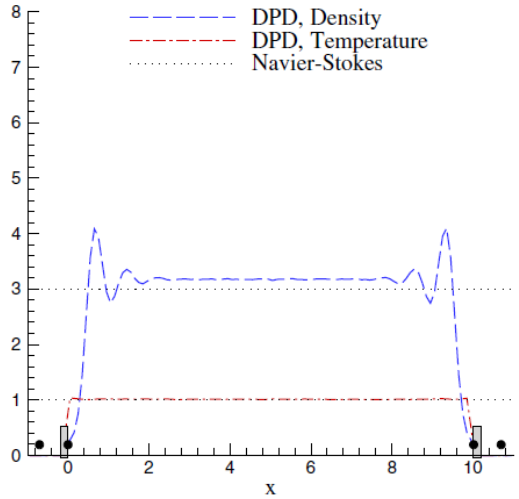

Figura 4.20: Densidade e Temperatura. (PIVKIN E KARNIADAKIS, 2005)

Como de praxe em trabalhos correlatos, uma das verificações é observar se os valores da temperatura e densidade obtidos condizem com os definidos para a simulação, assim observamos que a temperatura e densidade dos nossos resultados (figura 4.19) são condizentes aos resultados do trabalho de Karnadiakis (figura 4.20) e consequentemente os esperados para a simulação realizada.

A temperatura T de um sistema e sua energia cinética E_{kin} são relacionadas como (GRIEBEL *e outros*, 2007):

$$E_{kin} = \frac{3N}{2} K_B T \tag{4.2}$$

Desenvolvendo a expressão da equação 4.2

$$K_B T = \frac{2}{3N} E_{kin} \tag{4.3}$$

$$K_B T = \frac{2}{3N} \sum_{i=1}^{N} \frac{m_i}{2} {\mathbf{v}_i}^2 \tag{4.4}$$

$$K_B T = \frac{2}{3N} \sum_{i=1}^{N} \frac{m_i}{2} (v_{xi}^2 + v_{yi}^2 + v_{zi}^2) \tag{4.5}$$

Considerando as quantidades T^x e T^y como (REVENGA *e outros*, 1999), e estendendo para o caso 3D:

$$T^x = \sum_i m_i v_{xi}^2 \quad T^y = \sum_i m_i v_{yi}^2 \quad T^z = \sum_i m_i v_{zi}^2 \tag{4.6}$$

Nas quais as velocidades são medidas em relação a média do escoamento médio. Assim a temperatura é dada por:

$$T = \frac{1}{3} \left\langle T^x + T^y + T^z \right\rangle \tag{4.7}$$

Os valores são obtidos seccionando o canal e considerando na somatória da equação 4.6 somente as partículas que estão em cada seção. Na figura 4.21 apresentamos a evolução da temperatura cinética média média em função do número de iterações.

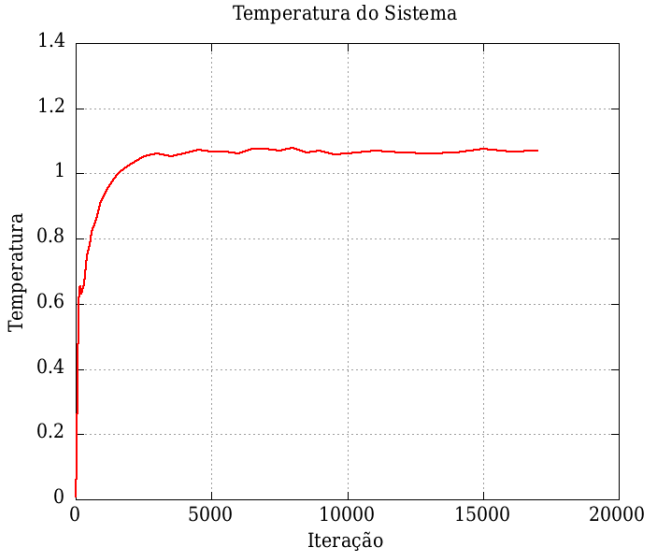

Figura 4.21: Temperatura do Sistema sem utilizar força externa

Observamos que este perfil de temperatura foi obtido sem a aplicação da força externa, neste caso o sistema entra em estado de equilíbrio por volta da iteração 5000.

A figura 4.22 apresenta a evolução da velocidade média máxima ao longo das iterações quando se aplica a força externa em todas as partículas do fluído, observamos que o estado de equilíbrio é atingido após a iteração 50000.

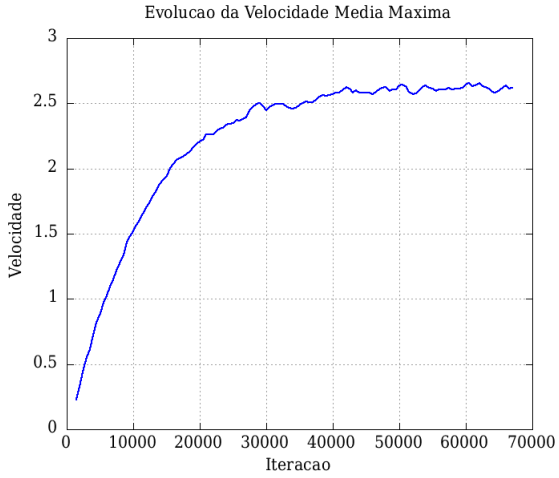

Figura 4.22: Evolução da velocidade média máxima ao longo das iterações, utilizando $f_{ext} = 0{,}01$

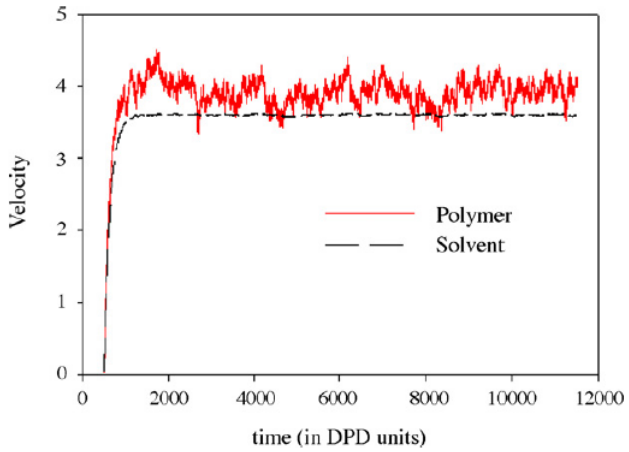

Figura 4.23: Evolução da velocidade (WANG *e outros*, 2008)

Em função do efeito termostático das componentes dissipativa e aleatória da força, a velocidade média máxima deve ao longo do tempo se estabilizar, o que podemos observar através da figura 4.22 em consonância com trabalho realizado por Wang *e outros* (2008) (figura 4.23).

Na figura 4.24 são apresentados os perfis de velocidade para as iterações de número 5.000, 9.000, 13.000, 17.000, 21.000, 25.000, 29.000 e 33.000 nas quais podemos observar a evolução do perfil de velocidade ao longo das iterações.

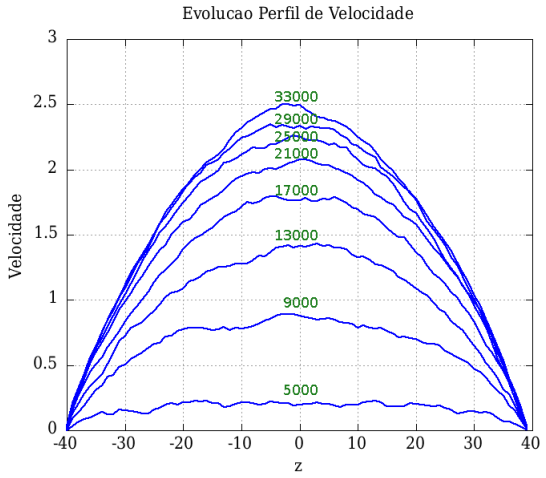

Figura 4.24: Evolução do perfil de velocidade, utilizando $f_{ext} = 0{,}01$

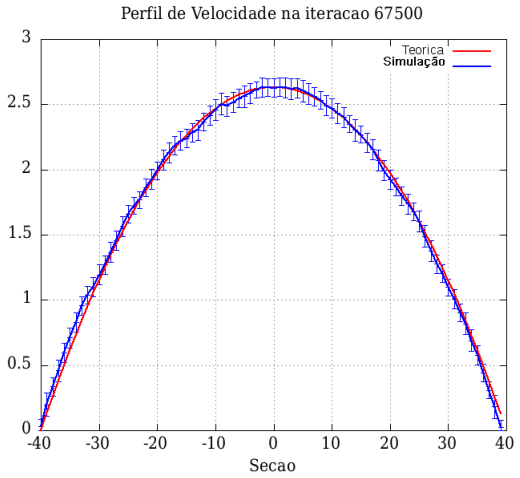

Figura 4.25: Perfil de velocidade na iteração 67500, utilizando $f_{ext} = 0{,}01$

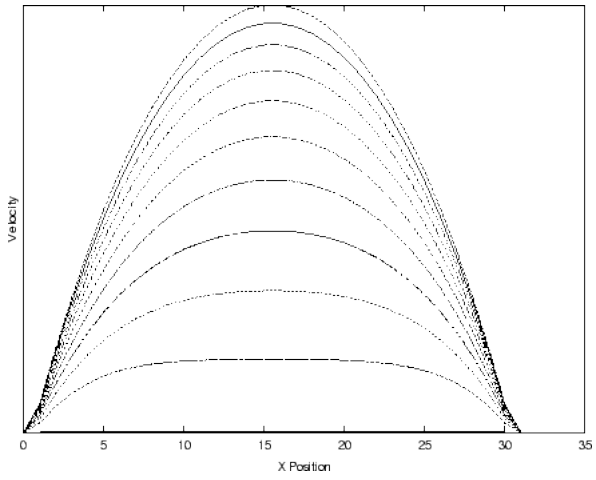

Figura 4.26: Evolução do perfil de velocidade obtido através software Lbm2d. `http://the.earth.li/~jon/work/lbetute/`, acessado em 10/11/2014

Na figura 4.25 é apresentado o perfil de velocidade para a iteração 67.500, na qual foram calculados os desvios padrão amostral para cada seção, considerando as 100 últimas iterações. Observamos que o resultado é coerente com o resultado analítico, sendo que o equilíbrio é atingido bem como o perfil de velocidade corresponde ao fluxo laminar condiz com o definido na teoria.

Comparando o exposto nas figuras 4.24 e 4.26 observamos que a evolução do perfil de velocidade no tempo é coerente com o proposto na literatura.

4.5 Tempos de Processamento

A tabela 4.7, apresenta para uma simulaçao com 17860 partículas os tempos em milisegundos a cada iteração, para cada uma das rotinas envolvidas, foi considerada nesta simulação a atualização da lista de vizinhos a cada 5 iterações. Para a rotina *Verlet2* os tempos foram menores que 1 milisegundo. Pelo exposto na tabela observamos que a construção da *Lista de Vizinhos* é responsável pela maior parte do processamento.

iteração	Lista de Células	Lista de Vizinhos	Verlet 1	Forças	Verlet 2
0	1	302	0	56	0
1	0	0	1	54	0
2	0	0	1	56	0
3	0	0	1	56	0
4	0	0	1	55	0
5	1	244	0	55	0
6	0	0	1	54	0
7	0	0	1	55	0
...					
10994	0	0	1	55	0
10995	1	245	0	55	0
10996	0	0	1	58	0
10997	0	0	1	55	0
10998	0	0	1	55	0
10999	0	0	1	57	0

Tabela 4.5: Tempos em milisegundos. Observar que 0ms pode significar que a rotina levou um tempo menor que 1ms para sua execução.

4.6 Simulação Realizada

As figuras 4.27 a 4.30 foram obtidas com o software de visualização VMD, e correspondem a alguns instantes da simulação. Para facilitar a visualização do movimento das partículas, parte das partículas correspondentes ao fluído foram coloridas de púrpura. Podemos perceber que em função da reflexão as partículas não adentram nas paredes. Podemos observar que é possível distinguir que o deslocamento das partículas é maior no centro do que nas proximidades das paredes.

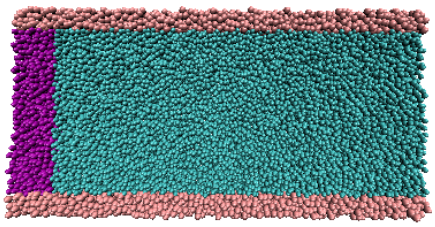

Figura 4.27: Iteração 0

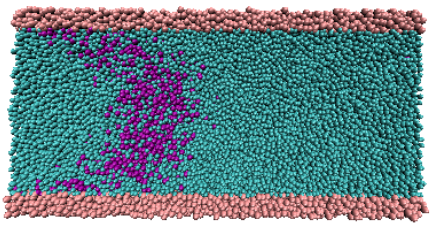

Figura 4.28: Iteração 3000

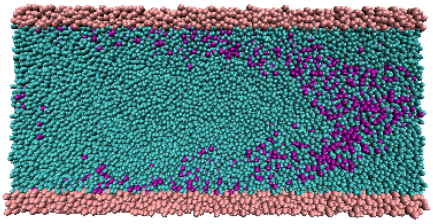

Figura 4.29: Iteração 5000

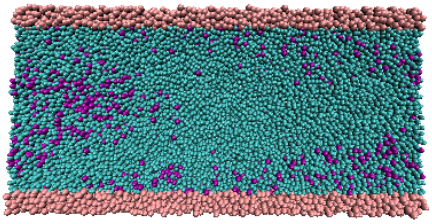

Figura 4.30: Iteração 6000

4.7 Acelerando o código

Os conceitos abordados e discutidos a seguir tem como objetivo indicar um possível caminho para melhoria do desempenho computacional do simulador, não sendo tratado em profundidade por não pertencer ao escopo do presente trabalho.

Os dados apresentados nesta seção foram obtidos pela execução do motor de simulação em

um computador com processador Intel I3 (4 threads [1]) com 4GBytes de memória, executando o sistema operacional linux, distribuição Ubuntu 10.04 LTS. Na simulação em questão foram utilizadas 17860 partículas representando uma fenda com 60rc x 3rc x 30rc.

Na simulação foram consideradas como periódicas as fronteiras nas direções x e y, os parâmetros utilizados são apresentados a seguir:

Parâmetro	Valor
delta_t	0.02
t_end	220
l	60, 3, 30
nc	40, 2, 20
r_cut	1,0
gamma	4,5
alfa	18.75
alfa1	8.66
sigma	3
f_cte	0.01, 0.0, 0.0

Tabela 4.6: Parâmetros utilizados

A tabela 4.7 apresenta o tempo gasto em milisegundos para a construção da lista de vizinhos em função ao número de *threads* utilizadas, são apresentados também o percentual de redução no tempo de execução, a aceleração obtida e a eficiência. Observamos que a partir de um determinado número de *threads* o aumento da quantidade de *threads* não corresponde a um ganho em termos de desempenho.

[1]Linha ou Encadeamento de execução, é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrencialmente. O suporte à thread é fornecido pelo próprio sistema operacional, ou implementada através de uma biblioteca de uma determinada linguagem. *wikipédia*

N Threads	Tempo(ms)	%	Speedup	Eficiência
1	211	1	1,00	1,00
2	157	25,59%	1,34	0,67
3	127	39,81%	1,66	0,51
4	122	42,18%	1,73	0,43
5	149	29,38%	1,42	0,28
6	138	34,60%	1,53	0,25
7	134	36,49%	1,57	0,22
8	134	36,49%	1,57	0,19

Tabela 4.7: Tempos em milisegundos

A figura 4.31, apresenta o tempo gasto em milisegundos para construir a lista de vizinhos utilizando diferentes números de threads, observamos que o ganho de tempo ocorre com o uso de até quatro threads, além disto o tempo de execução estabiliza-se.

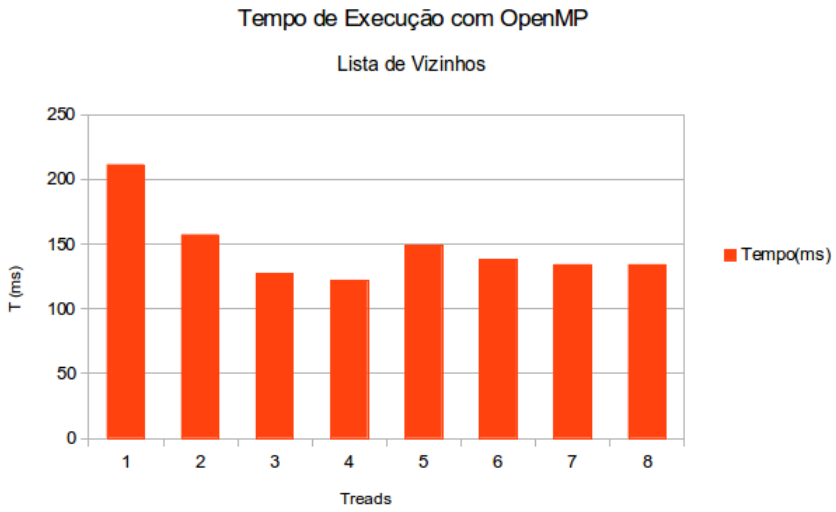

Figura 4.31: Tempo de execução

No gráfico 4.32 podemos observar que no computador em questão, a aceleração máxima é obtida com o uso de quatro threads, com seis ou mais threads a aceleração fica em torno de 60%.

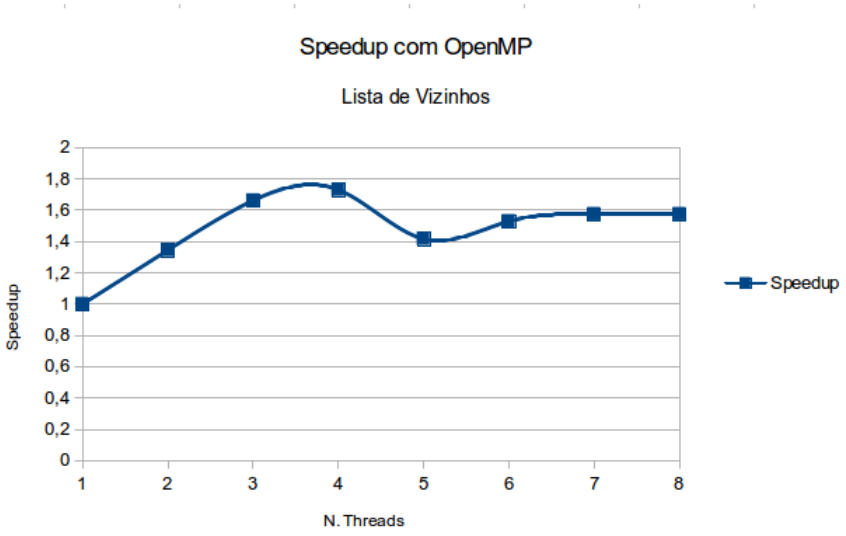

Figura 4.32: Ganho de desempenho com o uso do OpenMP

Pode-se concluir a partir da figura 4.32 que o máximo número de *threads* em que há ganho de desempenho é igual ao número de núcleos físicos do processador, quatro no equipamento utilizado.

Em função do que observamos nos gráficos anteriores, podemos observar no gráfico 4.33, que embora o aumento do número de threads possa representar um ganho em termos de tempo de execução, a medida com que aumentamos o número de threads a eficiência diminui, ou seja o ganho de tempo não é proporcional a quantidade de recursos utilizada.

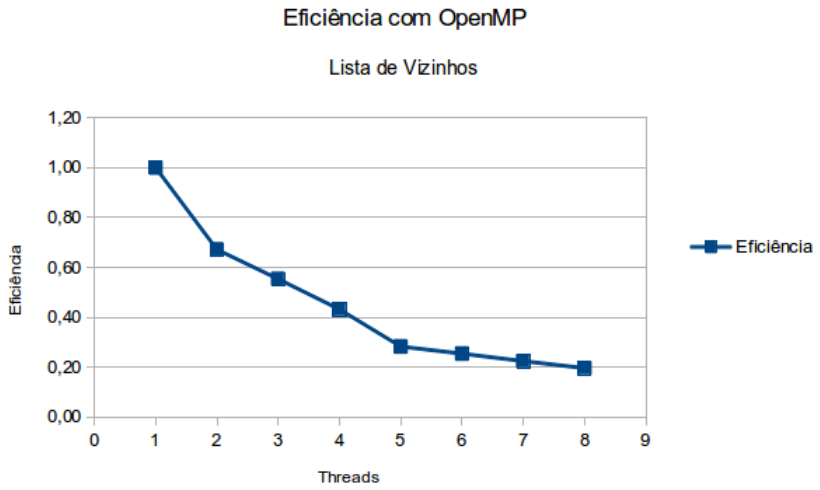

Figura 4.33: Eficiência obtida com o uso do OpenMP

5 Conclusão

Foi implementado um motor de simulação de fluidodinâmica usando o método DPD, programado em linguagem C/C++, utilizando as bibliotecas STL e OpenGL, composto das seguintes partes:

- Uma seção de entrada de dados, os quais são gerados por scripts escritos em linguagem Python;
- Um motor de simulação que utilizando o método DPD calcula a dinâmica das partículas de fluído;
- Um código 3D de visualização on-line da simulação, escrito em OpenGL;
- Uma interface para visualização em tempo real da simulação com o visualizador VMD;
- Um código para exportação dos dados da simulação para o programa Paraview;
- Um código para importação do arquivo XML gerados pelo simulador Hoomd-blue, o que permite que uma etapa da simulação seja gerado no simulador Hoomd-blue e em seguida continuada no simulador deste trabalho.

O código de entrada de dados permitiu que fossem gerados os arquivos de entrada das simulações, arquivos esses contendo as posições, velocidade, massa e tipo de cada partícula. Foram gerados arquivos para várias simulações utilizados com sucesso nas simulações.

O motor da simulação foi validado através de comparação com resultados da literatura e com aqueles obtidos através do simulador Hoomd-Blue. As partículas apresentaram comportamento dinâmico coerente com a mecânica de Newton. Foi possível induzir a vazão do conjunto de partículas dentro de volumes delimitados por fronteiras dos tipos periódica e *bounce-forward*, obtendo-se o comportamento coerente com o reportado na literatura.

Na maioria dos simuladores DPD há apenas a condição de fronteira periódica, o que é insuficiente para os problemas de engenharia, que demandam pelo menos a condição de fronteira de reflexão. Foi implementada a condição de fronteira do tipo "bounce-forward"no nosso motor de

simulação, de maneira a atender às necessidade da simulação de sistemas microfluídicos, apresentando resultados coerentes com a literatura.

Ao simular comportamento hidrodinâmico do fluxo em um tubo longo observou-se o perfil de velocidades de Haggen-Poiseuille.

A visualização on-line da simulação, implementada em OpenGL foi de grande importância para a depuração do motor da simulação. O código permite que o domínio da simulação seja rotacionado, aproximado ou afastado de acordo com a necessidade do usuário, exibindo as partículas de interesse com um código de cores correspondente a velocidade no principal eixo de deslocamento. Com esta finalidade os visualizadores com o VMD atendem não sendo necessário a implementação de um visualizador, contudo no que se refere aos aspectos de visualização on-line, vejo a possibilidade de que funcionalidades específicas sejam implementadas em nosso visualizador de maneira a suprir alguns aspectos que só estão disponíveis para visualização off-line nos visualizadores considerados.

A interface com o visualizador VMD funcionou perfeitamente permitindo uma maior flexibilidade na visualização on-line e off-line da simulação, em função do VMD oferecer uma enorme gama de recursos de visualização sendo otimizado para o uso de GPU.

O código de exportação para o Paraview permite uma análise melhor da simulação off-line pois possui uma série de ferramentas de análise de dados complexos tridimensionais.

O código de importação e exportação de arquivos XML permite a integração com o Hoomdblue. Seu uso foi imprescindível quando do inicio do desenvolvimento do simulador para que se tivesse a garantia de que a geometria proposta em um determinado problema fosse corretamente representada.

Visualizamos como trabalhos futuros:

- o aprimoramento da paralelização do código com o uso de OpenCL ou Cuda e OpenMPI
- melhor integração com a linguagem de script Python
- a incorporação de plugins, por exemplo:

        - no cálculo das forças o que permitirá o cálculo de outros tipos de interação
        - importação e exportação de dados
        - novos métodos de tratamento das fronteiras
        - métodos de integração
    - objetos compostos por várias partículas e quaternions
    - implementação de um mecanismo de parmetrização para auxiliar o usuário

O presente trabalho tem grande importância para o Laboratório de Supercomputação do DMC/FEM, pois servirá como ponto de partida para a construção de um framework de simulações em microflúidica com o Método DPD.

Referências

What is VMD? `http://www.ks.uiuc.edu/Research/vmd/allversions/what_is_vmd.html`, 2014. [Online; accessed 19-Dezembo-2013].

ALLEN, M.P. e TILDESLEY, D.J. **Computer simulation of liquids**. Clarendon Press, New York, NY, USA, 1989. ISBN 0-19-855645-4.

ANDERSEN, H.C. Molecular dynamics simulations at constant pressure and/or temperature. **The Journal of Chemical Physics**, v. 72, n. 4, 2384–2393, 1980.
URL: *http://scitation.aip.org/content/aip/journal/jcp/72/4/10.1063/1.439486*

ANDERSON, JOSHUA A., G.S.C. The development and expansion of hoomd-blue through six years of gpu proliferation. **ArXiv e-prints**, agosto 2013.

ANDERSON, J.; LORENZ, C. e TRAVESSET, A. General purpose molecular dynamics simulations fully implemented on graphics processing units. **Journal of Computational Physics**, v. 227, n. 10, 5342–5359, 2008.
URL: *http://linkinghub.elsevier.com/retrieve/pii/S0021999108000818*

BAYDÍM, Atilim Günes. **Dissipative Particle Dynamics and Coarse-Graining Review of Existing Techniques, Trials with Evolutionary Computation**. 2008. p. 109. Dissertação (Mestrado). Department of Applied Physics CHALMERS UNIVERSITY OF TECHNOLOGY.
URL: *http://www.chalmers.se/ap*

CHANDRA, R.; DAGUM, L.; KOHR, D.; MAYDAN, D.; MCDONALD, J. e MENON, R. **Parallel Programming in OpenMP**. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-671-8, 9781558606715.

DONADIO, M. How to generate white gaussian noise. http://www.dspguru.com/dsp/howtos/how-to-generate-white-gaussian-noise, novembro 2012. [Online; accessed 19-July-2008].

DUONG-HONG, D.; PHAN-THIEN, N. e FAN, X. An implementation of no-slip boundary conditions in dpd. **Computational Mechanics**, v. 35, n. 1, 24–29, 2004.
URL: *http://dx.doi.org/10.1007/s00466-004-0595-8*

ESPAÑOL, P. Dissipative particle dynamics. **EPL (Europhysics Letters)**, pp. 2503–25012, 2005.

ESPAÑOL, P. e WARREN, P. Statistical mechanics of dissipative particle dynamics. **EPL (Europhysics Letters)**, v. 30, n. 4, 191, 1995.
URL: *http://stacks.iop.org/0295-5075/30/i=4/a=001*

ESPAÑOL, P. Statistical mechanics of coarse-graining. In M. Karttunen; A. Lukkarinen e I. Vattulainen, editores, **Novel Methods in Soft Matter Simulations**, v. 640 de *Lecture Notes in Physics*, pp. 69–115. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-20916-4.
URL: *http://dx.doi.org/10.1007/978-3-540-39895-0_3*

FEDOSOV, D.A.; PIVKIN, I.V. e KARNIADAKIS, G.E. Velocity limit in {DPD} simulations of wall-bounded flows. **Journal of Computational Physics**, v. 227, n. 4, 2540 – 2559, 2008.
URL: *http://www.sciencedirect.com/science/article/pii/S0021999107004809*

FRENKEL, D. e SMIT, B., editores. **Understanding Molecular Simulation: From Algorithms to Applications**. Academic Press, Inc., Orlando, FL, USA, 1st ed., 1996. ISBN 0122673700.

FRENKEL, D. e SMIT, B. **Understanding Molecular Simulation: From Algorithms to Applications**. Computational science series. Elsevier Science, 2001. ISBN 9780080519982.
URL: *http://books.google.com.br/books?id=5qTzldS9ROIC*

GRIEBEL, M.; KNAPEK, S. e ZUMBUSCH, G. **Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications**. Springer Publishing Company, Incorpora-

ted, 1st ed., 2007. ISBN 3540680942, 9783540680949.

GROOT, R.D. e WARREN, P.B. Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation. **The Journal of Chemical Physics**, v. 107, n. 11, 4423–4435, 1997.
URL: *http://scitation.aip.org/content/aip/journal/jcp/107/11/10.1063/1.474784*

HOOGERBRUGGE, P.J. e KOELMAN, J.M.V.A. Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics. **Europhysics Letters (EPL)**, v. 19, n. 3, 155–160, junho 1992.
URL: *http://stacks.iop.org/0295-5075/19/i=3/a=001*

HORTA, A.A. e FILHO, R.M. Simulation of the alcohol-oil mixture in a t-shaped microchannel using the dissipative particle dynamics method on gpu devices. In **Proc. of the 14th Brazilian Congress of Thermal Sciences and Engineering, October 18-22, 2012, Rio de Janeiro, RJ, Brazil**. 2012.

HUMPHREY, W.; DALKE, A. e SCHULTEN, K. VMD – Visual Molecular Dynamics. **Journal of Molecular Graphics**, v. 14, 33–38, 1996.

JONATHAN E. MUELLER, D.F. e JACOB, T. **Electrocatalysis: Theoretical Foundations and Model Experiments**, v. 14 de *Advances in Electrochemical Sciences and Engineering*. Wiley-VCH Verlag GmbH & Co. KGaA, 2013. Chapter 1, Multiscale Modeling of Electrochemical Systems.

KNUTH, D.E. **The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-89684-2.

KOUMOUTSAKOS, P. Multiscale Flow Simulations using Particles. **Annual Review of Fluid Mechanics**, v. 37, n. 1, 457–487, 2005.
URL: *http://arjournals.annualreviews.org/doi/abs/10.1146/annurev.fluid.37.061903.175753*

LI, Z.; TANG, Y.H.; LEI, H.; CASWELL, B. e KARNIADAKIS, G.E. Energy-conserving dissipa-

tive particle dynamics with temperature-dependent properties. **Journal of Computational Physics**, v. 265, n. 0, 113 – 127, 2014.
URL: *http://www.sciencedirect.com/science/article/pii/S0021999114000977*

LIU, M.; MEAKIN, P. e HUANG, H. Dissipative particle dynamics simulation of fluid motion through an unsaturated fracture and fracture junction. **Journal of Computational Physics**, v. 222, n. 1, 110–130, março 2007.
URL: *http://linkinghub.elsevier.com/retrieve/pii/S0021999106003202*

MARTYS, N.S. e MOUNTAIN, R.D. Velocity verlet algorithm for dissipative-particle-dynamics-based models of suspensions. **Phys. Rev. E**, v. 59, 3733–3736, Mar 1999.
URL: *http://link.aps.org/doi/10.1103/PhysRevE.59.3733*

MATTSON, T.; SANDERS, B. e MASSINGILL, B. **Patterns for Parallel Programming**. Addison-Wesley Professional, first ed., 2004. ISBN 0321228111.

MEHBOUDI, A. e SAIDI, M. Physically based wall boundary condition for dissipative particle dynamics. **Microfluidics and Nanofluidics**, pp. 1–18, 2013.
URL: *http://dx.doi.org/10.1007/s10404-013-1285-7*

NIST/SEMATECH. e-handbook of statistical methods. April 2012.
URL: *http://www.itl.nist.gov/div898/handbook/*

PALMA, P.D.; VALENTINI, P.; NAPOLITANO, M. e DE PALMA, P. Dissipative particle dynamics simulation of a colloidal micropump. **Physics of Fluids**, v. 18, n. 2, 027103, 2006.
URL: *http://link.aip.org/link/PHFLE6/v18/i2/p027103/s1&Agg=doi*

PAN, D.; PHAN-THIEN, N.; MAI-DUY, N. e KHOO, B.C. Numerical investigations on the compressibility of a {DPD} fluid. **Journal of Computational Physics**, v. 242, n. 0, 196 – 210, 2013.
URL: *http://www.sciencedirect.com/science/article/pii/S002199911300123X*

PHILLIPS, C.L.; ANDERSON, J.A. e GLOTZER, S.C. Pseudo-random number generation for

Brownian Dynamics and Dissipative Particle Dynamics simulations on GPU devices. **Journal of Computational Physics**, v. 230, n. 19, 7191–7201, agosto 2011.

PIVKIN, I.V. e KARNIADAKIS, G.E. A new method to impose no-slip boundary conditions in dissipative particle dynamics. **Journal of Computational Physics**, v. 207, 114–128, 2005.

RANJITH, S.K.; PATNAIK, B.S.V. e VEDANTAM, S. No-slip boundary condition in finite-size dissipative particle dynamics. **J. Comput. Physics**, v. 232, n. 1, 174–188, 2013.

REVENGA, M.; ZÚÑIGA, I. e ESPAÑOL, P. Boundary conditions in dissipative particle dynamics. **Computer Physics Communications**, pp. 121–122, 1999.

REVENGA, M.; ZÚÑIGA, I.; ESPAÑOL, P. e PAGONABARRAGA, I. Boundary model in dpd. **International Journal of Modern Physics C**, 1998.

SCHILLER, Ulf Daniel. **Dissipative Particle Dynamics - A Study of the Methodological Background**. 2005. Dissertação (Mestrado). Faculty of Physics - University of Bielefeld.

SHREINER, D. **OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 3.0 and 3.1**. Pearson Education, Inc., Boston, MA, USA, 7rd ed., 2010. ISBN 978-0-321-5562-4.

STEINER, T. **Dissipative Particle Dynamics Simulation of Microfluidic Systems With Fluid Particle Methods on High Performance Computers**. Shaker Verlag, 2009.

STONE, J.E.; GULLINGSRUD, J. e SCHULTEN, K. A system for interactive molecular dynamics simulation. In **Proceedings of the 2001 Symposium on Interactive 3D Graphics**, I3D '01, pp. 191–194. ACM, New York, NY, USA, 2001. ISBN 1-58113-292-1.
URL: *http://doi.acm.org/10.1145/364338.364398*

THOMAS, D.B.; LUK, W.; LEONG, P.H. e VILLASENOR, J.D. Gaussian random number generators. **ACM Comput. Surv.**, v. 39, n. 4, novembro 2007.

URL: *http://doi.acm.org/10.1145/1287620.1287622*

VAN DER SPOEL, D.; VAN BUUREN, A.R.; APOL, E.; MEULENHOFF, P.J.; TIELEMAN, D.P.; SIJBERS, A.L.T.M.; HESS, B.; FEENSTRA, K.A.; LINDAHL, E.; VAN DRUNEN, R. e BERENDSEN, H.J.C. **Gromacs User Manual version 3.1**. Nijenborgh 4, 9747 AG Groningen, The Netherlands. Internet: http://www.gromacs.org, 2001.

VISSER, D.C.; HOEFSLOOT, H.C.J. e IEDEMA, P.D. Comprehensive boundary method for solid walls in dissipative particle dynamics. **J. Comput. Phys.**, v. 205, n. 2, 626–639, maio 2005.
URL: *http://dx.doi.org/10.1016/j.jcp.2004.11.020*

WANG, Y.; JIANG, W.; MILLER, S. e ECKSTEIN, E. Dissipative particle dynamics simulation of on-chip hydrodynamic chromatography. **Journal of Chromatography A**, v. 1198-1199, n. 0, 140 – 147, 2008.
URL: *http://www.sciencedirect.com/science/article/pii/S0021967308009321*

WILLEMSEN, S.M.; HOEFSLOOT, H.C.J. e IEDEMA, P.D. No-slip boundary condition in dissipative particle dynamics. **International Journal of Modern Physics C**, v. 11, n. 05, 881–890, 2000.
URL: *http://www.worldscientific.com/doi/abs/10.1142/S0129183100000778*

WRIGHT, R.; LIPCHAK, B. e HAEMEL, N. **Opengl® Superbible: Comprehensive Tutorial and Reference, Fourth Edition**. Addison-Wesley Professional, fourth ed., 2007. ISBN 9780321498823.

ZHANG, S.F.; XU, J.B.; WEN, H. e BHATTACHARJEE, S. Integration of rotational algorithms into dissipative particle dynamics: modeling polyaromatic hydrocarbons on the meso-scale. **Molecular Physics**, v. 109, n. 15, 1873–1888, August 2011.

A Tabela com Velocidades

A.1 Velocidades

A tabela a seguir apresenta para cada seção do duto a velocidade média obtida através da simulação e a velocidade teórica.

X	Vel. Simulação	Vel. Teórica
-42	-0,307902	0
-41	0,27878	0,083944807256236
-40	0,302788	0,165866848072563
-39	0,128492	0,24576612244898
-38	0,387249	0,323642630385487
-37	0,399421	0,399496371882086
-36	0,438622	0,473327346938776
-35	0,579374	0,545135555555556
-34	0,731168	0,614920997732426
-33	0,604839	0,682683673469388
-32	0,849525	0,74842358276644
-31	0,897512	0,812140725623583
-30	0,74888	0,873835102040816
-29	0,822769	0,933506712018141
-28	1,12381	0,991155555555556
-27	0,983706	1,04678163265306
-26	1,15698	1,10038494331066
-25	1,25778	1,15196548752834
-24	1,23097	1,20152326530612
-23	1,31832	1,24905827664399
-22	1,27119	1,29457052154195
-21	1,33573	1,33806
-20	1,42842	1,37952671201814
-19	1,32997	1,41897065759637
-18	1,67289	1,45639183673469
-17	1,44407	1,49179024943311
-16	1,63821	1,52516589569161

Tabela A.1: Velocidades parte 1

X	Vel. Simulação	Vel. Teórica
-15	1,33572	1,5565187755102
-14	1,45777	1,58584888888889
-13	1,55593	1,61315623582766
-12	1,62245	1,63844081632653
-11	1,60885	1,66170263038549
-10	1,62071	1,68294167800454
-9	1,56508	1,70215795918367
-8	1,55581	1,7193514739229
-7	1,61848	1,73452222222222
-6	1,78408	1,74767020408163
-5	1,59987	1,75879541950113
-4	1,68879	1,76789786848073
-3	1,71861	1,77497755102041
-2	1,69247	1,78003446712018
-1	1,75063	1,78306861678005
0	1,71637	1,78408
1	1,65345	1,78306861678005
2	1,74763	1,78003446712018
3	1,68859	1,77497755102041
4	1,70808	1,76789786848073
5	1,49087	1,75879541950113
6	1,7311	1,74767020408163
7	1,66193	1,73452222222222
8	1,60778	1,7193514739229
9	1,49922	1,70215795918367
10	1,5722	1,68294167800454
11	1,49591	1,66170263038549
12	1,52635	1,63844081632653

Tabela A.2: Velocidades parte 2

X	Vel. Simulação	Vel. Teórica
13	1,44181	1,61315623582766
14	1,54302	1,58584888888889
15	1,45641	1,5565187755102
16	1,40216	1,52516589569161
17	1,36544	1,49179024943311
18	1,398	1,45639183673469
19	1,4035	1,41897065759637
20	1,15108	1,37952671201814
21	1,30201	1,33806
22	1,10078	1,29457052154195
23	1,37772	1,24905827664399
24	1,25686	1,20152326530612
25	1,1467	1,15196548752834
26	1,14808	1,10038494331066
27	1,03699	1,04678163265306
28	0,932101	0,991155555555556
29	0,917071	0,933506712018141
30	0,780773	0,873835102040816
31	0,747929	0,812140725623583
32	0,675435	0,74842358276644
33	0,685066	0,682683673469388
34	0,554847	0,614920997732426
35	0,508289	0,545135555555556
36	0,47551	0,473327346938776
37	0,370824	0,399496371882086
38	0,235265	0,323642630385487
39	0,182472	0,24576612244898
40	0,139361	0,165866848072563
41	0,0889249	0,083944807256236
42	-0,0735477	0

Tabela A.3: Velocidades parte 3

B Setup da Simulação

B.1 Setup da Simulação

A seguir apresentamos o script em linguagem Python que gera um arquivo XML contendo a geometria da simulação, a qual é lida pelo motor de simulação.

Código B.1: Setup da Simulação

```
# -*- coding: utf-8 -*-
import random

# versao em uso que gera o aquivo xml no formato do Hoomd-Blue para ser
    utilizado com vmd

# Abstract struct class
class Struct:
    def __init__ (self, *argv, **argd):
        if len(argd):
            # Update by dictionary
            self.__dict__.update (argd)
        else:
            # Update by position
            attrs = filter (lambda x: x[0:2] != "__", dir(self))
            for n in range(len(argv)):
                setattr(self, attrs[n], argv[n])

# Specific class
class Particula(Struct):
    tag = 0
    massa = 0
    posicao = (0,0,0)
    velocidade = (0,0,0)
    tipo = ''

def grava_txt():
    f = open('workfile3', 'w')
        #f.write(('%d\n')%(N1+N2+N3+N4+N5))
        f.write(('%d\n')%(N+N5+N6+N7+N8))

```

```
31          for p in sistema:
32                  f.write('%c, %f, %f, %f, %f, %f, %f, %f\n'%(p.tipo, p.massa, p
                        .posicao[0], p.posicao[1], p.posicao[2], p.velocidade[0], p
                        .velocidade[1], p.velocidade[2]))
33          for p in sistema5:
34                  f.write('%c, %f, %f, %f, %f, %f, %f, %f\n'%(p.tipo, p.massa, p
                        .posicao[0], p.posicao[1], p.posicao[2], p.velocidade[0], p
                        .velocidade[1], p.velocidade[2]))
35          for p in sistema6:
36                  f.write('%c, %f, %f, %f, %f, %f, %f, %f\n'%(p.tipo, p.massa, p
                        .posicao[0], p.posicao[1], p.posicao[2], p.velocidade[0], p
                        .velocidade[1], p.velocidade[2]))
37          for p in sistema7:
38                  f.write('%c, %f, %f, %f, %f, %f, %f, %f\n'%(p.tipo, p.massa, p
                        .posicao[0], p.posicao[1], p.posicao[2], p.velocidade[0], p
                        .velocidade[1], p.velocidade[2]))
39          for p in sistema8:
40                  f.write('%c, %f, %f, %f, %f, %f, %f, %f\n'%(p.tipo, p.massa, p
                        .posicao[0], p.posicao[1], p.posicao[2], p.velocidade[0], p
                        .velocidade[1], p.velocidade[2]))
41          f.close()
42
43          return
44
45
46  mx = 30
47  my = 8
48  mz = 8
49
50  d = 4.0
51  rd = 1.0/(d**(1.0/3))
52
53  nx = int(mx/rd)
54  rdx = 1.0*mx/(nx-1)
55  #rdx = 1.0*mx/(nx)
56  ny = int(my/rd)
57  rdy = 1.0*my/(ny-1)
58  #rdy = 1.0*my/(ny)
59  nz = int(mz/rd)
60  rdz = 1.0*mz/(nz-1)
61  #rdz = 1.0*mz/(nz)
62
```

```
63  print d, mx, my, mz
64  print rd, nx, ny, nz
65  print rdx, rdy, rdz
66  print rdx*nx, rdy*ny, rdz*nz
67
68  N = nx*ny*nz
69  box = (mx, my, mz)
70
71  print N
72  print box
73
74  sistema = []
75
76  massa_cte = 1.0
77
78  for i in range(N):
79      p = Particula(tag=i)
80      sistema.append(p)
81
82  maxz = 0
83  minz = 999
84
85  maxy = 0
86  miny = 999
87
88  maxx = 0
89  minx = 999
90
91  for p in sistema:
92    (i, j, k) = (p.tag % nx, p.tag/nx % ny, p.tag/(nx*ny) % nz)
93    aux_x = i*rdx
94    aux_y = j*rdy
95    #aux_z = k*rdz+1
96    aux_z = k*rdz+0.375
97    p.posicao = (aux_x, aux_y, aux_z)
98    p.massa = massa_cte
99
100   minx = min(minx, aux_x)
101   maxx = max(maxx, aux_x)
102
103   miny = min(miny, aux_y)
104   maxy = max(maxy, aux_y)
```

```
105
106     minz = min(minz, aux_z)
107     maxz = max(maxz, aux_z)
108
109     # Particulas inicialmente sao do tipo A
110     p.tipo = '0'
111
112     # define uma fatia de particulas como sendo do tipo C
113     # para verificar o sentido do fluxo e se ele esta ocorrendo
114     if (k>=0 and k<=nz) and (j>=0 and j<=ny) and i>=0 and i<=3:
115       p.tipo = '2'
116
117 # inicializando as velocidades das particulas para
118 # tenham distribuicao normal
119 random.seed(1234)
120 T = 1.0
121 px = py = pz = 0.0
122 for p in sistema:
123       mass = p.massa
124       vx = random.gauss(0, T / mass)
125       vy = random.gauss(0, T / mass)
126       vz = random.gauss(0, T / mass)
127       p.velocidade = (vx, vy, vz)
128
129       # somando o momento total do sistema
130       px += mass*vx;
131       py += mass*vy;
132       pz += mass*vz;
133
134 # computando a media
135 px /= mx*my*mz;
136 py /= mx*my*mz;
137 pz /= mx*my*mz;
138
139 # subtraindo a velocidade media para que a energia do sistema seja zero
140 for p in sistema:
141       mass = p.massa;
142       v = p.velocidade;
143       # reduzindo a amplitude da aleatoriedade das particulas
144       p.velocity = ((v[0] - px/mass)*0.1, (v[1] - py/mass)*0.1, (v[2] - pz/mass)
              *0.1);
145
```

```
146 print miny, maxy
147 print minz, maxz
148
149 sistema5=[]
150 N5 = nx*ny
151
152 for i in range(N5):
153   p = Particula(tag=i)
154   sistema5.append(p)
155
156 for p in sistema5:
157   (i, j, k) = (p.tag % nx, p.tag/nx % ny, 0)
158   p.posicao = (rdx*i, rdy*j, minz-0.125)
159   p.massa = massa_cte
160   p.tipo = '1'
161   p.velocidade = (0.0, 0.0, 0.0)
162
163 sistema6=[]
164 N6 = (nx)*(ny)
165
166 for i in range(N6):
167   p = Particula(tag=i)
168   sistema6.append(p)
169
170 for p in sistema6:
171   (i, j, k) = (p.tag % (nx), p.tag/(nx) % (ny), 0)
172   p.posicao = (rdx*i+(rdx/2.0), rdy*j+(rdy/2.0), minz-0.375)
173   p.massa = massa_cte
174   p.tipo = '1'
175   p.velocidade = (0.0, 0.0, 0.0)
176
177 sistema7=[]
178 N7 = nx*ny
179
180 for i in range(N7):
181   p = Particula(tag=i)
182   sistema7.append(p)
183
184 for p in sistema7:
185   (i, j, k) = (p.tag % nx, p.tag/nx % ny, 0)
186   p.posicao = (rdx*i, rdy*j, maxz+0.125)
187   p.massa = massa_cte
```

```
188     p.tipo = '1'
189     p.velocidade = (0.0, 0.0, 0.0)
190
191 sistema8=[]
192 N8 = (nx)*(ny)
193
194 for i in range(N8):
195     p = Particula(tag=i)
196     sistema8.append(p)
197
198 for p in sistema8:
199     (i, j, k) = (p.tag % (nx), p.tag/(nx) % (ny), 0)
200     p.posicao = (rdx*i+(rdx/2.0), rdy*j+(rdy/2.0), maxz+0.375)
201     p.massa = massa_cte
202     p.tipo = '1'
203     p.velocidade = (0.0, 0.0, 0.0)
204
205 grava_txt()
206
207 #Geracao do arquivo xml
208 f = open('workfile3.xml', 'w')
209 #f.write(('%d\n')%(N+N1+N2+N3+N4+N5+N6+N7+N8))
210
211 f.write('<?xml version="1.0" encoding="UTF-8"?>\n')
212 f.write('<hoomd_xml version="1.4">\n')
213 f.write('<!-- this is a comment, you can put as many of these in the file\n')
214 f.write('     wherever you wish to. -->\n')
215 f.write('<configuration time_step="0">\n')
216 f.write('     <!-- data nodes go here -->\n')
217 f.write('     <box lx="%f" ly="%f" lz="%f"/>\n'%(maxx-minx+0.375, maxy-miny
        +0.375, maxz-minz+0.375))
218
219 f.write('     <position num="%d">\n'%(N))
220 for p in sistema:
221         f.write('         %f %f %f\n'%(p.posicao[0], p.posicao[1], p.posicao
                [2]))
222 for p in sistema5:
223         f.write('         %f %f %f\n'%(p.posicao[0], p.posicao[1], p.posicao
                [2]))
224 for p in sistema6:
225         f.write('         %f %f %f\n'%(p.posicao[0], p.posicao[1], p.posicao
                [2]))
```

```
226 for p in sistema7:
227         f.write('        %f %f %f\n'%(p.posicao[0], p.posicao[1], p.posicao
              [2]))
228 for p in sistema8:
229         f.write('        %f %f %f\n'%(p.posicao[0], p.posicao[1], p.posicao
              [2]))
230 f.write('    </position>\n')
231
232 f.write('    <velocity num="%d">\n'%(N))
233 for p in sistema:
234         f.write('        %f %f %f\n'%(p.velocidade[0], p.velocidade[1], p.
              velocidade[2]))
235 for p in sistema5:
236         f.write('        %f %f %f\n'%(p.velocidade[0], p.velocidade[1], p.
              velocidade[2]))
237 for p in sistema6:
238         f.write('        %f %f %f\n'%(p.velocidade[0], p.velocidade[1], p.
              velocidade[2]))
239 for p in sistema7:
240         f.write('        %f %f %f\n'%(p.velocidade[0], p.velocidade[1], p.
              velocidade[2]))
241 for p in sistema8:
242         f.write('        %f %f %f\n'%(p.velocidade[0], p.velocidade[1], p.
              velocidade[2]))
243 f.write('    </velocity>\n')
244
245 f.write('    <mass num="%d">\n'%(N))
246 for p in sistema:
247         f.write('        %f\n'%(p.massa))
248 for p in sistema5:
249         f.write('        %f\n'%(p.massa))
250 for p in sistema6:
251         f.write('        %f\n'%(p.massa))
252 for p in sistema7:
253         f.write('        %f\n'%(p.massa))
254 for p in sistema8:
255         f.write('        %f\n'%(p.massa))
256 f.write('    </mass>\n')
257
258 f.write('    <type num="%d">\n'%(N))
259 for p in sistema:
260         f.write('        %c\n'%(chr(ord(p.tipo)+17)))
```

```
261 for p in sistema5:
262         f.write('         %c\n'%(chr(ord(p.tipo)+17)))
263 for p in sistema6:
264         f.write('         %c\n'%(chr(ord(p.tipo)+17)))
265 for p in sistema7:
266         f.write('         %c\n'%(chr(ord(p.tipo)+17)))
267 for p in sistema8:
268         f.write('         %c\n'%(chr(ord(p.tipo)+17)))
269 f.write('     </type>\n')
270
271 f.write('</configuration>\n')
272 f.write('</hoomd_xml>\n')
273
274 f.close()
```

C Código Fonte do Motor de Simulação

C.1 Código Fonte do Motor de Simulação

A seguir é apresentado o código do motor de simulação implementado em Linguagem C++.

Código C.1: Código fonte do motor de simulação

```
1  // g++ msdpd.cc -o msdpd imd/imd.cpp imd/vmdsock.cpp -fopenmp `pkg-config --
       cflags --libs libxml++-2.6`
2  // g++ msdpd.cc -o msdpd  -fopenmp `pkg-config --cflags --libs libxml++-2.6`
3
4  #include <iostream>
5  #include <fstream> // fstream
6  #include <stdlib.h>
7
8  #include <libxml++/libxml++.h>
9  #include <libxml++/parsers/textreader.h>
10
11 #include <cstring>       /*strtok strcpy*/
12 #include <math.h>        /* floor */
13
14 #include <boost/timer.hpp>
15 #include "boost/date_time/posix_time/posix_time.hpp"
16
17 #define index(ic,nc) ((ic)[0] + (nc)[0]*((ic)[1] + (nc)[1]*(ic)[2]))
18 #define sqr(x) ((x)*(x))
19 #define max(a,b) ((a)<(b))?(b):(a)
20 //#define abs(a) ((a)<0)?-(a):(a)
21
22 #define Vwall -0.5
23
24 typedef double real;
25
26 typedef boost::posix_time::ptime Time;
27 typedef boost::posix_time::time_duration TimeDuration;
28
29 boost::timer timer;
30
31 Time t_inicial[20];
```

```
32 long tempo[10];
33
34 /**
35  * @author Agnus A. Horta
36  * @version 1.0
37  *
38  * <Uma curta linha de descricao>
39  *
40  * <Descricao maior>
41  * <Pode criar multiplas linhas ou paragrafos se quiser>
42  *
43  * @param  Descricao dos parametros de entrada de funcoes ou metodos
44  * @param  ...
45  * @return Descricao do valor de retorno
46  */
47
48 /**
49  * @author Agnus A. Horta
50  * @version 1.0
51  *
52  * Log do tempo de execucao das rotinas
53  *
54  * <Descricao maior>
55  * <Pode criar multiplas linhas ou paragrafos se quiser>
56  *
57  * @param  rotina :  codigo da rotina para a qual esta sendo calculado o tempo
        de execucao
58  * @param  start : se verdadeiro inicia a contagem do tempo, caso contrario
       encerra a contagem
59  * @return t  : time step
60  */
61 void msglog(int rotina, int start, long t)
62 {
63     //static time_t s;
64     //static Time t1;
65     static std::ofstream arq1;
66     std::string str = "temp_";
67     std::ostringstream convert;
68
69     if(rotina==-1)
70     {
71         convert << t;
```

```
72          str += convert.str();
73          str += ".csv";
74          arq1.open (str.c_str());
75          arq1 << "iteracao" << "\t" <<  "Lista de Celulas" << "\t" << "Lista de
                Vizinhos" << "\t" << "Verlet 1" << "\t" << "Forcas" << "\t" << "
               Verlet 2" << std::endl;
76      }
77      else if ( rotina==0 )
78      {
79          arq1 << t << "\t" <<  tempo[1] << "\t" << tempo[2] << "\t" <<tempo[3]
               << "\t" <<tempo[4] << "\t" <<tempo[5] << std::endl;
80          for(int i=0; i<=5; i++) tempo[i]=0;
81      }
82      else if ( rotina==-2 )
83      {
84          arq1.close();
85      }
86      else if (start)
87      {
88          t_inicial[rotina] = boost::posix_time::microsec_clock::local_time();
89      }
90      else
91      {
92          Time t1 = t_inicial[rotina];
93          Time t2(boost::posix_time::microsec_clock::local_time());
94          TimeDuration dt = t2 - t1;
95          tempo[rotina] = dt.total_milliseconds();
96      }
97  }
98
99  // Baseado no codigo proposto por:
100 // Dmitry Aleks
101 // http://stackoverflow.com/questions/1970698/c-malloc-for-two-dimensional-
        array
102 int** dAlloc(int rows, int columns)
103 {
104
105     int header = rows * sizeof(int*);
106     int body = rows*columns*sizeof(int);
107     int** rowptr = (int**)malloc(header + body);
108     int* buf  = (int*)(rowptr + rows);
109     rowptr[0] = buf;
```

```
110     for (int k = 1; k < rows; ++k)
111     {
112         rowptr[k] = rowptr[k-1] + columns;
113     }
114     return rowptr;
115
116 }
117
118 //
119 // Generate a random number between 0 and 1
120 // return a uniform number in [0,1].
121 double uniform()
122 {
123     return rand() / double(RAND_MAX);
124 }
125
126 /**/
127 /* How to Generate White Gaussian Noise */
128 /* by Matt Donadio */
129 /* http://www.dspguru.com/dsp/howtos/how-to-generate-white-gaussian-noise */
130
131 float cltm_rand_gen(float mean, float variance)
132 {
133     float x = 0, u;
134     int N = 25;
135
136     for (int i = 1; i <= N; i++)
137     {
138         u = uniform();
139         x += u;
140     }
141
142     /* for uniform randoms in [0,1], mu = 0.5 and var = 1/12 */
143     /* adjust X so mu = 0 and var = 1 */
144
145     x = x - N / 2.0; /* set mean to 0 */
146     x = x * sqrt(12.0 / N); /* adjust variance to 1 */
147
148     return mean + sqrt(variance) * x;
149 }
150
151
```

```
152 class SistemaParticula
153 {
154     int qt;
155     real *r;
156     real *v;
157     real *m;
158     real *f;
159     int *ti;
160 
161     int *ll;
162     int *head;
163 
164     int *NN;
165     int **NBL;
166 
167     real t;
168     real dt;
169 
170     int use_vmd;
171     real t_end;
172     int nc[3];
173     int pnc;
174     real l[3];
175     real rc;
176     real sigma;
177     real epsilon;
178     real gm;
179     real al;
180     real all;
181     real sg;
182     real fb[3];
183     float max_dist;
184     real vm[100];
185     real vmy[100];
186     real vmz[100];
187     real vm2[100];
188     real vmx2, vmy2, vmz2;
189     int cv[100];
190 
191     void inicializa(int);
192     void ll_inicializa(void);
193 
```

```
194     void NBL_inicializa(int, int);
195     void import_3f(std::string, int qt, real *);
196     void import_1f(std::string, int qt, real *);
197     void import_1i(std::string, int *);
198     void Verlet1(int);
199     void forcas(void);
200     void forca_LJ(void);
201     void Verlet2(void);
202 
203     real minimum_image(int, int, real *);
204 
205 public:
206     int xml_import(char *);
207     void parametros_le(char *);
208     void parametros_exibe(void);
209     void grava_xml(void);
210     void ll_constroi(void);
211     void ll_mostra(void);
212     void NBL_constroi(void);
213     void NBL_mostra(void);
214     void velocidade(void);
215     void integracao(void);
216     void warm_up(float);
217     void grava_vel_med(void);
218 
219 };
220 
221 real SistemaParticula::minimum_image(int i, int j, real * rv)
222 {
223     real dr = 0, dx;
224 
225     dx = r[i*3]-r[j*3];
226     if (abs(dx)>(l[0]*0.5)) dx=dx-copysign(l[0], dx); //sgn(dx)*l[d];
227     rv[0] = dx;
228     dr += sqr(dx);
229 
230     dx = r[i*3+1]-r[j*3+1];
231     if (abs(dx)>(l[1]*0.5)) dx=dx-copysign(l[1], dx); //sgn(dx)*l[d];
232     rv[1] = dx;
233     dr += sqr(dx);
234 
235     dx = r[i*3+2]-r[j*3+2];
```

```
236     if (abs(dx)>(l[2]*0.5)) dx=dx-copysign(l[2], dx); //sgn(dx)*l[d];
237     rv[2] = dx;
238     dr += sqr(dx);
239
240     rv[3] = dr;
241
242     return dr;
243
244 }
245
246 void SistemaParticula::inicializa (int q)
247 {
248
249     qt = q;
250     r = (real *) malloc(sizeof(real)*3*qt);
251     v = (real *) malloc(sizeof(real)*3*qt);
252     m = (real *) malloc(sizeof(real)*qt);
253     f = (real *) calloc(3*qt, sizeof(real));
254     ti = (int *) malloc(sizeof(real)*qt);
255     t = 0.0;
256     dt = 1.0e-2;
257
258     for (int i=0; i<=100; i++)
259     {
260         vm[i]=0.0;
261         vm2[i]=0.0;
262         vmy[i]=0.0;
263         vmz[i]=0.0;
264         cv[i]=0;
265     }
266
267     vmx2=0.0;
268     vmy2=0.0;
269     vmz2=0.0;
270
271     if (r==NULL || v==NULL || m==NULL || f==NULL || ti==NULL)
272     {
273         std::cerr << "Nao foi possivel alocar memoria!" << std::endl;
274         exit(1);
275     }
276
277 }
```

```
278
279 void SistemaParticula::import_3f(std::string values, int qt, real * d)
280 {
281
282     int pos1,pos2;
283     real x, y, z;
284     std::string s1   = "";
285
286     s1.assign(values);
287     char * pEnd;
288     x = strtod (s1.c_str(),&pEnd);
289     y = strtod (pEnd,&pEnd);
290     z = strtod (pEnd,&pEnd);
291
292     d[0] = x;
293     d[1] = y;
294     d[2] = z;
295
296     for (int i=1; i<qt; i++)
297     {
298
299         x = strtod (pEnd,&pEnd);
300         y = strtod (pEnd,&pEnd);
301         z = strtod (pEnd,&pEnd);
302
303         d[i*3] = x;
304         d[i*3+1] = y;
305         d[i*3+2] = z;
306
307     }
308
309 }
310
311 void SistemaParticula::import_1f(std::string values, int qt, real * d)
312 {
313
314     int pos1,pos2;
315     real x, y, z;
316     std::string s1   = "";
317
318     s1.assign(values);
319     char * pEnd;
```

```
320
321     x = strtod (s1.c_str(),&pEnd);
322     d[0] = x;
323
324     for (int i=1; i<qt; i++)
325     {
326
327         x = strtod (pEnd,&pEnd);
328         d[i] = x;
329
330     }
331
332 }
333
334 void SistemaParticula::import_1i(std::string values, int * d)
335 {
336
337     int x;
338     int p=0, i=0;
339     while (p<values.size())
340     {
341         if (values[p]>='A' && values[p]<='Z')
342         {
343             x = values[p]-'A';
344             d[i++] = x;
345         }
346         p++;
347     }
348 }
349
350 int SistemaParticula::xml_import(char * arq)
351 {
352     // Set the global C and C++ locale to the user-configured locale,
353     // so we can use std::cout with UTF-8, via Glib::ustring, without
        exceptions.
354     //std::locale::global(std::locale(""));
355     int ch=0, ch1=0;
356     std::string s2   = "";
357
358     try
359     {
360         xmlpp::TextReader reader(arq);
```

```
361
362         while (reader.read())
363         {
364             int depth = reader.get_depth();
365
366             if (ch==0)
367             {
368                 if ( reader.get_name()=="position" )
369                     ch=1;
370                 else if ( reader.get_name()=="velocity" )
371                     ch=2;
372                 else if ( reader.get_name()=="mass" )
373                     ch=3;
374                 else if ( reader.get_name()=="type" )
375                     ch=4;
376             }
377             else if (ch<0) ch=0;
378
379             if (reader.has_attributes())
380             {
381                 reader.move_to_first_attribute();
382                 do
383                 {
384                     if (ch==1 && reader.get_name()=="num")
385                     {
386                         qt = atoi(reader.get_value().c_str());
387                         SistemaParticula::inicializa(qt);
388                     }
389                 }
390                 while (reader.move_to_next_attribute());
391                 reader.move_to_element();
392             }
393
394             if (reader.has_value())
395                 if (ch==1)
396                 {
397                     ch=-1;
398                     SistemaParticula::import_3f(reader.get_value(), qt, r );
399                 }
400                 else if (ch==2)
401                 {
402                     ch=-1;
```

```
403                     SistemaParticula::import_3f(reader.get_value(), qt, v );
404                 }
405                 else if (ch==3)
406                 {
407                     ch=-1;
408                     SistemaParticula::import_1f(reader.get_value(),qt, m);
409                 }
410                 else if (ch==4)
411                 {
412                     ch=-1;
413                     SistemaParticula::import_1i(reader.get_value(), ti);
414                 }
415         }
416 
417 
418         for(int i=0; i<qt; i++)
419         {
420             r[i*3] += 30.0;
421             r[i*3+1] += 1.5;
422             r[i*3+2] += 15.0;
423         }
424 
425     }
426     catch (const std::exception& e)
427     {
428         std::cerr << "Exception caught: " << e.what() << std::endl;
429         return EXIT_FAILURE;
430     }
431 
432     return EXIT_SUCCESS;
433 
434 }
435 
436 void SistemaParticula::grava_vel_med(void)
437 {
438     std::ofstream arq;
439     std::string str = "vel_";
440     std::ostringstream convert;
441     real dv;
442     convert << ((int)(t/dt));
443     str += convert.str();
444     str += ".csv";
```

```
445     arq.open (str.c_str());
446
447     arq << vmx2 << "\t" << vmy2 << "\t" << vmz2 << "\n";
448     for (int i=0; i<=100; i++)
449     {
450         if(cv[i]<1) dv=-1;
451         else dv = sqrt( ( vm2[i]-(1/cv[i])*sqr(vm[i]) )/(cv[i]-1) );
452         arq << (i-50)<< "\t" << (vm[i]/cv[i]) << "\t" << (vmy[i]/cv[i]) << "\t
               " << (vmz[i]/cv[i]) << "\t" << cv[i] << "\t" << dv << "\n ";
453     }
454
455     arq.close();
456
457
458 }
459
460 void SistemaParticula::grava_xml()
461 {
462     std::ofstream arq;
463     std::string str = "dump_";
464     std::ostringstream convert;
465     convert << ((int)(t/dt));
466     str += convert.str();
467     str += ".xml";
468     arq.open (str.c_str());
469
470
471     arq << "<?xml version='1.0' encoding='UTF-8'?>\n";
472     arq << "<hoomd_xml version='1.4'>\n";
473     arq << "<!-- Arquivo gerado com o motor de simulacao do laboratorio de
           microfluidica\n";
474     arq << "    DMC/DPM/FEM. -->\n";
475     arq << "<configuration time_step='" << t << "'>\n";
476     arq << "    <!-- data nodes go here -->\n";
477     //arq << "    <box lx='"<< l[0]<<"' ly='"<<l[1]<<"' lz='"<<l[2]<<"'/>\n";
478
479     arq << "    <position num='" << qt << "'>\n";
480     for (int i=0; i<qt; i++) arq << "        " << r[i*3] << " " << r[i*3+1] <<
           " " << r[i*3+2] << "\n";
481     arq << "    </position>\n";
482
483     arq << "    <velocity num='" << qt << "'>\n";
```

```
484     for (int i=0; i<qt; i++) arq << "         "<< v[i*3] << " " << v[i*3+1] <<
            " " << v[i*3+2] << "\n";
485     arq << "    </velocity>\n";
486
487     arq << "    <mass num='" << qt << "'>\n";
488     for (int i=0; i<qt; i++) arq << "         " << m[i]<< "\n";
489     arq << "    </mass>\n";
490
491     arq << "    <type num='" << qt << "'>\n";
492     for (int i=0; i<qt; i++) arq << "         " << (char)(ti[i]+65) << "\n";
493     arq << "    </type>\n";
494
495     arq << "</configuration>\n";
496     arq << "</hoomd_xml>\n";
497
498     arq.close();
499 }
500
501 void SistemaParticula::parametros_le(char * arq)
502 {
503
504     std::ifstream inputdata;
505     std::string line;
506     std::vector<real> param;
507
508     char *pch;
509
510     inputdata.open (arq);
511     if (inputdata.is_open())
512     {
513         int i=0;
514         while ( inputdata.good() )
515         {
516             std::getline( inputdata, line );
517             char * cstr = new char [line.length()+1];
518             std::strcpy (cstr, line.c_str());
519             pch = std::strtok(cstr," ");
520             while (pch != NULL)
521             {
522                 param.push_back(atof(pch));
523                 pch = std::strtok (NULL, " ");
524             }
```

```
525             if (0==i) use_vmd = param[0];
526             else if (1==i) dt = param[0];
527             else if (2==i) t_end = param[0];
528             else if (3==i)
529             {
530                 l[0] = param[0];
531                 l[1] = param[1];
532                 l[2] = param[2];
533             }
534             else if (4==i) rc = param[0];
535             else if (5==i) sigma = param[0];
536             else if (6==i) epsilon = param[0];
537             else if (7==i) gm = param[0];
538             else if (8==i) al = param[0];
539             else if (9==i) al1 = param[0];
540             else if (10==i) sg = param[0];
541             else if (11==i)
542             {
543                 fb[0] = param[0];
544                 fb[1] = param[1];
545                 fb[2] = param[2];
546             }
547 
548             i++;
549             param.clear();
550 
551             SistemaParticula::ll_inicializa();
552             SistemaParticula::NBL_inicializa(1000,qt);
553 
554         }
555 
556     }
557     else std::cout << "Nao foi possivel abrir o arquivo de parametros." << std
        ::endl;
558 
559 }
560 
561 void SistemaParticula::parametros_exibe()
562 {
563 
564     std::cout << "use_vmd = " << use_vmd << std::endl;
565     std::cout << "delta_t = " << dt << std::endl;
```

```
566     std::cout << "t_end = " << t_end << std::endl;
567     std::cout << "l = " << l[0] << ", " << l[1] << ", " << l[2] << std::endl;
568     std::cout << "nc = " << nc[0] << ", " << nc[1] << ", " << nc[2] << std::
            endl;
569
570     std::cout << "r_cut = " << rc << std::endl;
571     std::cout << "Lennard Jones" << std::endl;
572     std::cout << "   sigma = " << sigma << std::endl;
573     std::cout << "   epsilon = " << epsilon << std::endl;
574     std::cout << "DPD" << std::endl;
575     std::cout << "   gamma = " << gm << std::endl;
576     std::cout << "   alfa = " << al << std::endl;
577     std::cout << "   alfa1 = " << al1 << std::endl;
578     std::cout << "   sigma = " << sg << std::endl;
579     std::cout << "fcte = " << fb[0] << ", " << fb[1] << ", " << fb[2] << std::
            endl;
580 }
581
582 void SistemaParticula::ll_mostra(void)
583 {
584
585     for (int i=0; i<pnc; i++)
586     {
587         if (head[i]>-1)
588         {
589             std::cout << "Celula " << i << " contem as particulas:\n\t";
590             int k;
591             k = head[i];
592             while (k>=0)
593             {
594                 std::cout << k << ", ";
595                 k = ll[k];
596             }
597             std::cout << std::endl;
598         }
599     }
600
601 }
602
603 void SistemaParticula::ll_constroi(void)
604 {
605
```

```
606     int kc[3];
607
608     //cell grid
609
610     for (int i=0; i<pnc; i++) head[i] = -1;
611
612     for (int i=0; i<qt; i++)
613     {
614
615         int ind;
616
617         // treat the periodic boundary conditions PBC
618         if ( r[i*3]<0 ) r[i*3] +=l[0];
619         else if ( r[i*3]>l[0] ) r[i*3] -=l[0];
620
621         if ( r[i*3+1]<0 ) r[i*3+1] +=l[1];
622         else if ( r[i*3+1]>l[1] ) r[i*3+1] -=l[1];
623
624         if ( r[i*3+2]<0 ) r[i*3+2] +=l[2];
625         else if ( r[i*3+2]>l[2] ) r[i*3+2] -=l[2];
626
627         kc[0] = (int)floor(r[i*3]*nc[0] / l[0]);
628         kc[1] = (int)floor(r[i*3+1]*nc[1] / l[1]);
629         kc[2] = (int)floor(r[i*3+2]*nc[2] / l[2]);
630
631         ind = index(kc,nc);
632
633         ll[i] = head[ind];
634         head[ind] = i;
635
636     }
637
638 }
639
640 void SistemaParticula::ll_inicializa()
641 {
642
643     nc[0]=floor(l[0]/(rc*1.5)); // eq 3.33
644     nc[1]=floor(l[1]/(rc*1.5));
645     nc[2]=floor(l[2]/(rc*1.5)); // eq 3.33
646
647     pnc = nc[0]*nc[1]*nc[2];
```

```
648
649     head = (int *) calloc(pnc, sizeof(int));
650     ll = (int *) calloc(qt, sizeof(int));
651
652 }
653
654 void SistemaParticula::NBL_constroi(void)
655 {
656     //  Neighbor list
657     int ic[3],  kcx[3];
658     int kc[3];
659     real rd[4];
660     //int c = 0;
661
662     #pragma omp parallel for
663     for (int i=0; i<qt; i++) NN[i] = 0;
664
665     for (int i=0; i<qt; i++)
666     {
667
668         ic[0] = (int)floor(r[i*3]*nc[0] / l[0]);
669         ic[1] = (int)floor(r[i*3+1]*nc[1] / l[1]);
670         ic[2] = (int)floor(r[i*3+2]*nc[2] / l[2]);
671
672         for (kc[0]=ic[0]-1; kc[0]<=ic[0]+1; kc[0]++)
673         {
674             for (kc[1]=ic[1]-1; kc[1]<=ic[1]+1; kc[1]++)
675             {
676                 for (kc[2]=ic[2]-1; kc[2]<=ic[2]+1; kc[2]++)
677                 {
678
679                     // treat the periodic boundary conditions
680                     for (int l=0; l<3; l++)
681                         if (kc[l]<0) kcx[l] = nc[l]+kc[l];
682                         else if (kc[l]>=nc[l]) kcx[l] = kc[l]%nc[l];
683                         else kcx[l] = kc[l];
684
685                     int ind = index(kcx,nc);
686                     int k = head[ind];
687
688                     while ( k>=0 )
689                     {
```

```
690                             real dr = 0;
691                             dr = SistemaParticula::minimum_image(i, k, rd);
692                             if (i!=k)
693                             {
694 
695                                 if (dr<=sqr(rc*1.5))
696                                 {
697                                     if (i<k)
698                                     {
699                                         NBL[NN[i]][i] = k;
700                                         NN[i]++;
701                                     }
702                                 }
703                             }
704 
705                             k=ll[k]; // proxima particula da celula
706                         }
707                     }
708                 }
709             }
710 
711         }
712 
713 }
714 
715 void SistemaParticula::NBL_inicializa(int rows, int cols)
716 {
717     NBL = dAlloc(rows, cols);
718     NN = (int *) calloc(qt, sizeof(int));
719 }
720 
721 void SistemaParticula::NBL_mostra()
722 {
723 
724     std::cout << "lista de Vizinhos\n";
725     for (int i=0; i<qt; i++)
726     {
727         std::cout << "particula " << i << " tem " << NN[i] << " vizinhos" <<
                std::endl;
728         for (int j=0; j<NN[i]; j++)
729             std::cout << "NBL(" << j << "," << i << ")= " << NBL[j][i] << std
                    ::endl;
```

```
730         }
731     }
732
733     void SistemaParticula::Verlet1(int bf)
734     {
735
736         real ax, ay, az;
737         real vx, vy, vz;
738         real dx, dy, dz;
739         real dt2 = sqr(dt);
740
741         float c0=1.0f, c1=1.0f, c2=0.5f;
742
743         max_dist = 0.0f;
744
745         for (int i=0; i<qt; i++)
746         {
747             if(ti[i]!=1)
748             {
749                 ax = f[i*3]/m[i];
750                 ay = f[i*3+1]/m[i];
751                 az = f[i*3+2]/m[i];
752                 dx = c1*v[i*3]*dt+c2*ax*dt2;
753                 dy = c1*v[i*3+1]*dt+c2*ay*dt2;
754                 dz = c1*v[i*3+2]*dt+c2*az*dt2;
755                 vx = c0*v[i*3]+(c1-c2)*ax*dt;
756                 vy = c0*v[i*3+1]+(c1-c2)*ay*dt;
757                 vz = c0*v[i*3+2]+(c1-c2)*az*dt;
758                 max_dist = max(max_dist, sqr(dx)+sqr(dy)+sqr(dz));
759                 r[i*3]+=dx;
760                 r[i*3+1]+=dy;
761                 r[i*3+2]+=dz;
762                 v[i*3] = vx;
763                 v[i*3+1] = vy;
764                 v[i*3+2] = vz;
765
766                 // bouncy-forward
767                 if(bf==1)
768                 {
769                     if(r[i*3+2]<3.0)
770                     {
771                         real dr=3.0-r[i*3+2];
```

```
772                         r[i*3+2]+=2*dr;
773                         v[i*3] = -vx;
774                         v[i*3+1] = -vy;
775                         v[i*3+2] = -vz;
776                     }
777                     else if(r[i*3+2]>(l[2]-3.0))
778                     {
779                         real dr=r[i*3+2]-(l[2]-3.0);
780                         r[i*3+2]-=2*dr;
781                         v[i*3] = -vx;
782                         v[i*3+1] = -vy;
783                         v[i*3+2] = -vz;
784                     }
785                 }
786             }
787         }
788 
789         max_dist = sqrt(max_dist);
790     }
791 
792     void SistemaParticula::Verlet2(void)
793     {
794 
795         float c2 = 0.5f;
796         for(int i=0; i<qt; i++)
797         {
798             if(ti[i]!=1)
799             {
800                 v[i*3]+=c2*f[i*3]*dt;
801                 v[i*3+1]+=c2*f[i*3+1]*dt;
802                 v[i*3+2]+=c2*f[i*3+2]*dt;
803             }
804         }
805     }
806 
807     void SistemaParticula::forcas(void)
808     {
809         // Calcula as forcas
810 
811         //Require: Fk is initialized to 0 for all k
812         //Require: NBLji only stores neighbors where i < k
813 
```

```
814     real rd[4];
815     real vi[3], vk[3];
816 
817     #pragma omp parallel for
818     for (int i=0; i<qt; i++)
819     {
820         if (ti[i]!=1)
821         {
822             f[i*3]=fb[0];
823             f[i*3+1]=fb[1];
824             f[i*3+2]=fb[2];
825         }
826 
827     }
828 
829     for (int i=0; i<qt; i++)
830     {
831         for (int j=0; j<NN[i]; j++)
832         {
833             int k = NBL[j][i];
834             real dr = 0;
835             dr = SistemaParticula::minimum_image(i, k, rd);
836             //std::cout << "-> " << i << " " << k << " "<< dr << std::endl;
837 
838             if (dr<=sqr(rc))
839             {
840 
841                 /* Lennard-Jones */
842                 float sr = (sigma/sqrt(dr));
843                 float sr2 = sqr(sr);
844                 float sr6 = sqr(sr2*sr);
845                 float sr12 = sr6*sr6;
846 
847                 float fl = 4*epsilon*(sr12-sr6);
848 
849                 /*Forca conservativa*/
850 
851                 float mag = sqrt(rd[3]);
852                 float aux = 1 / mag;
853                 float aij = al;
854 
855                 if ( ti[k]==1 || ti[i]==1 ) aij = al1; // utiliza um
```

```
856
857             real e0 = rd[0] * aux;
858             real e1 = rd[1] * aux;
859             real e2 = rd[2] * aux;
860
861             float w = (1 - mag / rc);
862             float fc = (aij * w);
863
864             /*Forca dissipativa*/
865             float g = gm; /*amplitude da repulsao para a forca dissipativa
                    */
866
867             float vx, vy, vz;
868             /*
869                     if(ti[i]==1) {
870                         vi[0] = -v[k*3];
871                         vi[1] = -v[k*3+1];
872                         vi[2] = -v[k*3+2];
873                     } else {
874                         vi[0] = v[i*3];
875                         vi[1] = v[i*3+1];
876                         vi[2] = v[i*3+2];
877                     }
878
879                     if(ti[k]==1) {
880                         vk[0] = -v[i*3];
881                         vk[1] = -v[i*3+1];
882                         vk[2] = -v[i*3+2];
883                     } else {
884                         vk[0] = v[k*3];
885                         vk[1] = v[k*3+1];
886                         vk[2] = v[k*3+2];
887                     }
888             */
889
890             vi[0] = v[i*3];
891             vi[1] = v[i*3+1];
892             vi[2] = v[i*3+2];
893
894             vk[0] = v[k*3];
```

```
895                     vk[1] = v[k*3+1];
896                     vk[2] = v[k*3+2];
897
898                     vx = vi[0] - vk[0];
899                     vy = vi[1] - vk[1];
900                     vz = vi[2] - vk[2];
901
902                     float dot = e0 * vx + e1 * vy + e2 * vz;
903                     float fd = ((-g * sqr(w)) * dot);
904
905                     /*Forca aleatoria*/
906                     //float fr = sg * w * cltm_rand_gen(0.0,1.0)*sqrt(dt);
907                     float fr = sg * w * (2*uniform()-1)/sqrt(dt);
908
909                     float fx = e0*(fc+fd+fr); //+fb[0];
910                     float fy = e1*(fc+fd+fr); //+fb[1];
911                     float fz = e2*(fc+fd+fr); //+fb[2];
912
913
914                     //std::cout << "i=" << i << " k=" << k << " r=" << r << " f=("
                            << f[0] << ", " << f[1] << ", " << f[2] << ")" << std::
                           endl;
915
916                     if (ti[i]!=1)
917                     {
918                         f[i*3]+=fx;
919                         f[i*3+1]+=fy;
920                         f[i*3+2]+=fz;
921                     }
922
923                     if (ti[k]!=1)
924                     {
925                         f[k*3]-=fx;
926                         f[k*3+1]-=fy;
927                         f[k*3+2]-=fz;
928                     }
929                 }
930             }
931         }
932 }
933
934 void SistemaParticula::forca_LJ(void)
```

```
935 {
936     // Calcula as forcas
937
938     //Require: Fk is initialized to 0 for all k
939     //Require: NBLji only stores neighbors where i < k
940
941     real rd[4];
942
943     #pragma omp parallel for
944     for (int i=0; i<qt; i++)
945     {
946         f[i*3]=0;
947         f[i*3+1]=0;
948         f[i*3+2]=0;
949     }
950
951     for (int i=0; i<qt; i++)
952     {
953         //if (ti[i]!=1) { // como nao esta considerando a 3a. lei de Newton,
                nao precisa computar as forcas para a fp congelada
954         for (int j=0; j<NN[i]; j++)
955         {
956             int k = NBL[j][i];
957             real dr = 0;
958             dr = SistemaParticula::minimum_image(i, k, rd);
959
960             if (dr<=sqr(rc))
961             {
962
963                 /* Lennard-Jones */
964                 float sr = (sigma/sqrt(dr));
965                 float sr2 = sqr(sr);
966                 float sr6 = sqr(sr2)*sr2;
967                 float sr12 = sr6*sr6;
968
969                 float fl = 4*epsilon*(sr12-sr6);
970
971                 float mag = sqrt(rd[3]);
972                 float aux = 1 / mag;
973
974                 real e0 = rd[0] * aux;
975                 real e1 = rd[1] * aux;
```

```
976                     real e2 = rd[2] * aux;
977
978                     float fx = e0*fl;
979                     float fy = e1*fl;
980                     float fz = e2*fl;
981
982                     if (ti[i]!=1)
983                     {
984                         f[i*3]+=fx;
985                         f[i*3+1]+=fy;
986                         f[i*3+2]+=fz;
987                     }
988
989                     if (ti[k]!=1)
990                     {
991                         f[k*3]-=fx;
992                         f[k*3+1]-=fy;
993                         f[k*3+2]-=fz;
994                     }
995                 }
996             }
997         }
998 }
999
1000 void SistemaParticula::velocidade(void)
1001 {
1002     real vx_max=0,vy_max=0,vz_max=0, vel_med=0;
1003     int ct, id;
1004
1005     for (int i=0; i<qt; i++)
1006     {
1007         if(ti[i]!=1)
1008         {
1009             if((t/dt)>9499)
1010             {
1011
1012                 vmx2+=sqr(v[i*3]);
1013                 vmy2+=sqr(v[i*3+1]);
1014                 vmz2+=sqr(v[i*3+2]);
1015
1016                 if(r[i*3+2]>0 && r[i*3+2]<=l[2])
1017                 {
```

```
1018                        id = (int)floor(r[i*3+2]/l[2]*100);
1019                        vm[id]+=v[i*3];
1020                        vmy[id]+=v[i*3+1];
1021                        vmz[id]+=v[i*3+2];
1022                        vm2[id]+=sqr(v[i*3]);
1023                        cv[id]++;
1024                    }
1025                }
1026                if(v[i*3]>vx_max) vx_max=v[i*3];
1027                if(v[i*3+1]>vy_max) vy_max=v[i*3+1];
1028                if(v[i*3+2]>vz_max) vz_max=v[i*3+2];
1029                vel_med += v[i*3];
1030                ct++;
1031            }
1032        }
1033        vel_med/=ct;
1034        std::cout << "Iteracao : " << (int)(t/dt) << "(" << t << ")    \tVx_max = "
             << vx_max << "\tVy_max = " << vy_max << "\tVz_max = " << vz_max<< "
            \tVel_med = " << vel_med << std::endl;
1035    }
1036
1037    void SistemaParticula::integracao(void)
1038    {
1039
1040        int ct=0;
1041        int k=250;
1042
1043        while(t<t_end)
1044        {
1045
1046            if((ct%5)==0)
1047            {
1048                msglog(1,1,0);
1049                SistemaParticula::ll_constroi();
1050                msglog(1,0,0);
1051                msglog(2,1,0);
1052                SistemaParticula::NBL_constroi();
1053                msglog(2,0,0);
1054                ct = 0;
1055            }
1056            ct++;
1057            msglog(3,1,0);
```

```
1058            SistemaParticula::Verlet1(1);
1059            msglog(3,0,0);
1060
1061            if((t/dt)>=9400) SistemaParticula::velocidade();
1062            else std::cout << (int)(t/dt) << std::endl;
1063            msglog(4,1,0);
1064            SistemaParticula::forcas();
1065            msglog(4,0,0);
1066            msglog(5,1,0);
1067            SistemaParticula::Verlet2();
1068            msglog(5,0,0);
1069            msglog(0,0,(int)(t/dt));
1070            t+=dt;
1071            if (((int)(t/dt))%k==0)
1072            {
1073                SistemaParticula::grava_xml();
1074                if((t/dt)>=9500)
1075                {
1076                    SistemaParticula::grava_vel_med();
1077                    k=50;
1078                }
1079            }
1080        }
1081
1082    }
1083
1084    void SistemaParticula::warm_up(float tend)
1085    {
1086
1087        while(t<tend)
1088        {
1089
1090            SistemaParticula::ll_constroi();
1091            SistemaParticula::NBL_constroi();
1092            SistemaParticula::Verlet1(0);
1093            SistemaParticula::forca_LJ();
1094            SistemaParticula::Verlet2();
1095
1096            t+=dt;
1097        }
1098
1099        for(int i=0; i<qt; i++)
```

```
1100        {
1101            if( r[i*3+2]<=3.0 || r[i*3+2]>=(l[2]-3.0) ) ti[i]=1;
1102            else if ( r[i*3]<4.0 ) ti[i]=2;
1103
1104            v[i*3] = 0.0;
1105            v[i*3+1] = 0.0;
1106            v[i*3+2] = 0.0;
1107
1108        }
1109
1110        t=0;
1111        SistemaParticula::grava_xml();
1112
1113    }
1114
1115    int main(int argc, char** argv)
1116    {
1117        int qt;
1118        SistemaParticula sp;
1119
1120        std::cout << "Arquivo de dados: " << argv[1] << std::endl;
1121
1122        sp.xml_import(argv[1]);
1123        sp.grava_xml();
1124
1125        sp.parametros_le(argv[2]);
1126        sp.parametros_exibe();
1127
1128        //sp.warm_up(1.0);
1129        //sp.ll_constroi();
1130        //sp.ll_mostra();
1131        //sp.NBL_constroi();
1132        //sp.NBL_mostra();
1133        msglog(-1,0,0); // inicializa a contagem do tempo
1134        sp.integracao();
1135        msglog(-2,0,0);
1136        sp.grava_vel_med();
1137
1138        return 0;
1139
1140    }
```

D Arquivo de Dados Geométricos

D.1 Arquivo XML

Apresentamos a seguir um exemplo de um arquivo XML gerado pelo script em Python. Este formato foi baseado no formato utilizado no simulador Hoomd-Blue.

```
<?xml version='1.0' encoding='UTF-8'?>
<hoomd_xml version='1.4'>
<!-- Arquivo gerado com o motor de simulação do laboratório de microfluídica
    DMC/DPM/FEM. -->
<configuration time_step='220'>
    <!-- data nodes go here -->
    <position num='17860'>
        1.38553 1.10631 27.0722
        1.52598 1.87033 29.0824
        .
        .
        .
        5.74042 2.59075 16.1551
    </position>
    <velocity num='17860'>
        0 0 0
        0 0 0
        .
        .
        .
        0 0 0
   </velocity>
   <mass num='17860'>
        1
        1
        .
        .
        .
        1
    </mass>
    <type num='17860'>
```

```
        B
        B
        .
        .
        .
        A
    </type>
    </configuration>
</hoomd_xml>
```

APÊNDICE A – Discretização no Tempo

A.1 Processo de discretização

O processo de discretização descreve a transição de um problema que está definido em um intervalo contínuo para um conjunto finito de pontos. Inicialmente utilizada na solução de equações diferenciais transformando-as em um sistema de equações cuja solução se aproxima da solução das equações diferenciais nos pontos de discretização. O uso da discretização simplifica o processo de se calcular os novos parâmetros das partículas em função dos anteriores, e as forças correspondentes (GRIEBEL *e outros*, 2007).

A.1.1 Fórmula básica de discretização

O intervalo de tempo $[0,t_{final}] \subset \mathbb{R}$, no qual o sistema de equações diferenciais será resolvido, é dividido em l subintervalos de mesmo tamanho, $\delta t = t_{final}/l$. Desta maneira cria-se um grid que contém os pontos $t_n = n.\delta t, n = 0, \cdots l$, localizados no final de cada subintervalo. A equação diferencial será considerada apenas nestes pontos no tempo. De acordo com a definição da derivada:

$$\frac{dx}{dt} = \lim_{\delta t \to 0} \frac{x(t+\delta t) - x(t)}{\delta t} \tag{A.1}$$

da função diferenciável $x : \mathbb{R} \to \mathbb{R}$ o operador diferencial dx/dt pode ser aproximado no ponto t_n do grid de pontos pelo operador de diferenças

$$\left[\frac{dx}{dt}\right]_n^r = \frac{x(t_{n+1}) - x(t_n)}{\delta t} \tag{A.2}$$

Sendo $t_{b+1} = t_n + \delta t$ o próximo ponto no grid a direita de t_n. A expansão por Taylor da função x no ponto t_{n+1} será

$$x(t_n + \delta t) = x(t_n) + \delta t \frac{dx}{dt}(t_n) + O(\delta t^2) \tag{A.3}$$

o que implica em um erro da ordem de $O(\delta t)$ para a aproximação da primeira derivada. Portanto, se o intervalo de tempo é dividido pela metade, espera-se que o erro causado pela discretização no tempo também caia para a metade. O operador diferencial dx/dt no ponto t_n do grid poderá ser aproximado pelo operador de diferença central

$$\left[\frac{dx}{dt}\right]_n^c = \frac{x(t_{n+1}) - x(t_{n-1})}{2\delta t} \tag{A.4}$$

No caso da aproximação da primeira deriva por série de Taylor obtemos um erro da ordem de $O(\delta t^2)$. Para a segunda derivada d^2x/dt^2 será aproximada pelo operador de diferença

$$\left[\frac{d^2x}{dt^2}\right]_n = \frac{1}{\delta t^2}(x(t_n + \delta t) - 2x(t_n) + x(t_n - \delta t)) \tag{A.5}$$

Pela expansão de Taylor até a terceira ordem em torno dos pontos $t_n + \delta t$ e $t_n - \delta t$ obtemos

$$x(t_n + \delta t) = x(t_n) + \delta t \frac{dx(t_n)}{dt} + \frac{1}{2}\delta t^2 \frac{d^2x(t_n)}{dt^2} + \frac{1}{6}\delta t^3 \frac{d^3x(t_n)}{dt^3} + O(\delta t^4) \tag{A.6}$$

e

$$x(t_n + \delta t) = x(t_n) - \delta t \frac{dx(t_n)}{dt} + \frac{1}{2}\delta t^2 \frac{d^2x(t_n)}{dt^2} - \frac{1}{6}\delta t^3 \frac{d^3x(t_n)}{dt^3} + O(\delta t^4) \tag{A.7}$$

substituindo estas expansões na Equação (A.5) obtemos

$$\left[\frac{d^2x}{dt^2}\right]_n = \frac{d^2x(t_n)}{dt^2} + O(\delta t^2) \tag{A.8}$$

Portanto o erro de discretização desta aproximação da derivada segunda por Equação (A.5) também é da ordem $O(\delta t^2)$.

A.1.2 Algoritmo de Verlet

O algoritmo de Verlet, sem utilizar da velocidade da partícula, calcula a posição da partícula para o instante $(t + \Delta t)$ com base na posição atual e anterior Steiner (2009).

$$\mathbf{r}(t + \Delta t) = 2\mathbf{r}(t) - \mathbf{r}(t - \Delta t) + \Delta t^2\mathbf{a}(t) \tag{A.9}$$

Este método é de fácil implementação sendo suficientemente estável para intervalos de tempos moderados, tendo como desvantagem o fato da velocidade não estar explicita nos cálculos, e sob certas condições torna-se numericamente instável. No caso da velocidade ser necessária, ela poderá ser aproximada pela posição $\mathbf{r}$ da partícula nos instantes de tempo $(t - \Delta t)$ e $(t + \Delta t)$ (STEINER, 2009)

$$\mathbf{v}(t) = \frac{\mathbf{r}(t + \Delta t) - \mathbf{r}(t - \Delta t)}{2\Delta t} \tag{A.10}$$

A.1.3 Algoritmo Leap-Frog

O algoritmo Leap-Frog é uma variação do algoritmo de Verlet com a vantagem de que a velocidade está explícita no algoritmo. (STEINER, 2009)

$$\begin{aligned} \mathbf{r}(t+\Delta t) &= \mathbf{r}(t) + \Delta t \mathbf{v}(t+\tfrac{1}{2}\Delta t) \\ \mathbf{v}(t+\tfrac{1}{2}\Delta t) &= \mathbf{v}(t-\tfrac{1}{2}\Delta t) + \Delta t \mathbf{a}(t) \end{aligned} \tag{A.11}$$

A velocidade no instante de tempo t pode ser calculada como a média entre as velocidades nos instantes $(t+\frac{1}{2}\Delta t)$ e $(t-\frac{1}{2}\Delta t)$:

$$\mathbf{v(t)} = \frac{1}{2}\left[\mathbf{v}(t+\frac{1}{2}\Delta t) + \mathbf{v}(t-\frac{1}{2}\Delta t)\right] \tag{A.12}$$

O inconveniente deste método é que as posições e velocidades ainda são calculadas em instantes de tempo diferentes (STEINER, 2009).

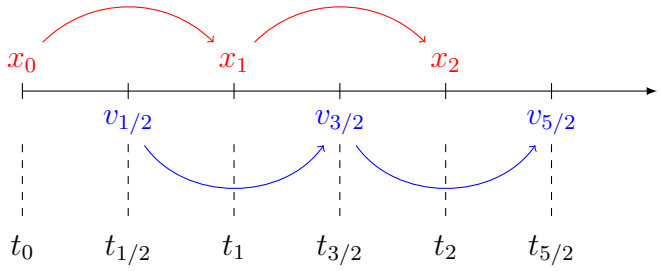

Figura A.1: Integração Leap-Frog.